Aprendendo Django no Planeta Terra - vol. 2

Marinho Brandão

1ª edição - 2009

revisão

Mychell Neiva Rodrigues

ilustrações e capa

João Matheus Mazzia de Oliveira

impressão e acabamento

Lulu.com

E-mail: marinho@aprendendodjango.com

www.aprendendodjango.com

Licença

O autor desta obra e de seus personagens é **José Mário Neiva Brandão** (codinome **"Marinho Brandão"**).

A autoria das ilustrações é de **João Matheus Mazzia de Oliveira**.

A revisão e edição do texto é de **Mychell Neiva Rodrigues**.

Sumário

Volume 2

Volume 1

Volume 2

Dicas de Aprendizado

Antes de entrar de cabeça na leitura do livro, é bom manter em mente alguns princípios que vão facilitar a compreensão, manter a calma em alguns momentos e fortalecer o aprendizado.

Veja abaixo:

Não copie e cole, digite

A pior coisa a fazer para se aprender algo é buscar o atalho de copiar e colar, com aquela velha auto-ilusão "ahh, já entendi isso, vou copiar e colar só pra confirmar que eu entendi".

Isso não *cola* - ou pelo menos não cola na sua memória. Digite cada comando, cada linha, cada coisinha, faça por você mesmo, e os resultados serão mais sólidos e duradouros.

Calma, uma coisa de cada vez

O detalhamento inicial é feito assim por sua própria natureza. À medida que algumas coisas são explicadas detalhadamente, elas passam a ser apontadas de forma mais objetiva dali em diante, e coisas em foco naquele capítulo passam a ser mais detalhadas.

Assim, se você continua pensando que o livro está detalhando demais, é um bom sinal de que aprendeu as lições dos capítulos anteriores de tal forma que nem notou. Parabéns!

Não existe mágica, não se iluda com isso

O Django não é uma ferramenta mágica. Muito menos de truques.

A produtividade do Django está ligada a três coisas muito importantes:

- **Framework.** Trabalhar com um framework é mais produtivo simplesmente porque você não faz coisas que já estão prontas e levaram

anos para serem criadas de forma que você possa usar facilmente agora. Esse valor só tem essa força e dinâmica porque o Django é **software livre**.

- **Conhecimento.** O seu conhecimento do Django e de conceitos aprofundados de **Web**, **Programação Orientada a Objetos** e **Bancos de Dados** faz TODA a diferença.

- **Python.** É uma linguagem de fácil compreensão, não burocrática e prática. Muito prática. Isso evita que seu cérebro se perca em interpretar coisas e símbolos que não precisaria interpretar e mantenha o foco na solução.

Portanto, não existem atalhos, nem vida fácil, mas você pode facilitar as coisas e se divertir com isso, usando as ferramentas certas, e agindo com foco e persistência. Então, numa bela manhã vai perceber que está produzindo com qualidade e rapidez, por seus próprios méritos.

Versão do Django

A versão do Django adotada nesta série é a **versão 1.0.2**.

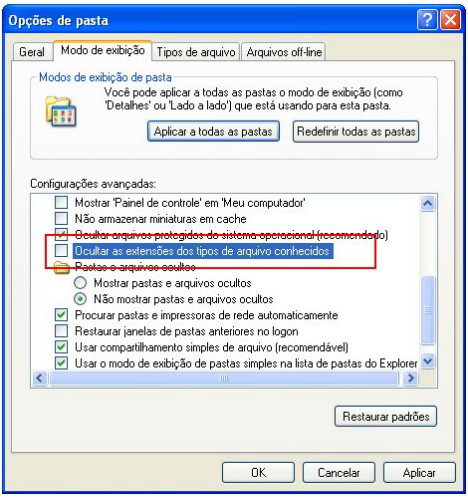
Algumas pessoas têm encontrado algumas dificuldades em decorrência de usarem uma versão diferente. Há diferenças entre versões em aspectos que temos trabalhado, portanto, use a versão **1.0.2 ou superior**.

Fique ligado nas extensões do arquivos

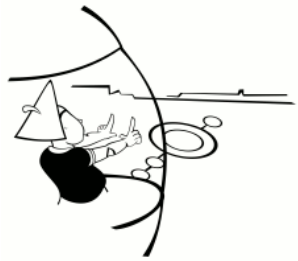
No **Windows**, o padrão é ocultar as extensões dos arquivos. Por exemplo: para um arquivo **"teste.txt"**, será exibido apenas como **"teste"**.

Isso é tremendamente chato para o desenvolvedor.

Para facilitar a sua vida, vá numa janela do **Explorer**, ao menu **Ferramentas** -> **Opções de Pasta** e desabilite a opção que você ve na imagem abaixo.



Capítulo 19: Signals e URLs amigáveis com Slugs



Em Katara há uma classe de artistas auto-denominados **"Artistas Peônicos Geométricos Biocorretistas Surfadores"**, um nome criado por eles que não é visto da mesma forma pelos agricultores.

Sua arte consiste em levar seus discos voadores até uma plantação bacana, amarrar um cabo à nave, e prender um cortador de grama na outra extremidade do cabo.

Surfando em pranchas voadoras, eles giram suas naves enquando sobem e descem no ar, cortando e quebrando a plantação em formas geométricas como bolas, bolinhas, bolonas, meias-luas e outras formas curvas sem nenhum fundamento. Há aqueles de nível avançado que já fazem quadradinhos, losangos e outras coisas, mas ainda assim geométricas demais para agradar a qualquer **crítico de arte** ou **agricultor**.

Depois de um tempo, esses Artistas Peônicos Geométricos Biocorretistas Surfadores foram expulsos de Katara, pra deixar seus críticos de arte e agricultores em paz.

E saíram em busca de outros planetas. Mas agora eles gostam de dar um clima à situação, antes de se revelar como autores da arte.

Alguns agricultores de outros planetas já começaram a reclamar, mas os críticos de arte ainda não se pronunciaram a respeito, devido a não saber exatamente quem fez e assim ficam com medo de ser alguém famoso e descolado.

O que eles ainda não notaram é alguns agricultores entenderam sua arte como **sinais do além**, e a cada novo desenho que é feito numa plantação, seus novos **seguidores** reagem aos "sinais" de forma cada vez mais convencida.

Para desenhos com bolinhas eles dançam uma dança especial. Para desenhos com quadradinhos soltam fogos de artifício. E assim por diante.

Usando signals para criar slugs

Antes de falar em **Signals** vamos falar dos **Slugs**.

"Slug" é uma expressão que não tem nada a ver com uma **lesmas**. "Slug" é uma expressão do meio jornalístico para criar identificações mais claras e intuitivas para conteúdo publicado na web.

Dessa forma, o monte bolinhas, quadradinhos e risquinhos abaixo não é visto exatamente como uma obra de arte:

```
http://turismo.terra.com.br/interna/0,,OI1768089-EI176,00.html
```

Pois ela poderia ser assim:

```
http://turismo.terra.com.br/interna/confira-dicas-para-arrumar-malas-para-viagem/
```

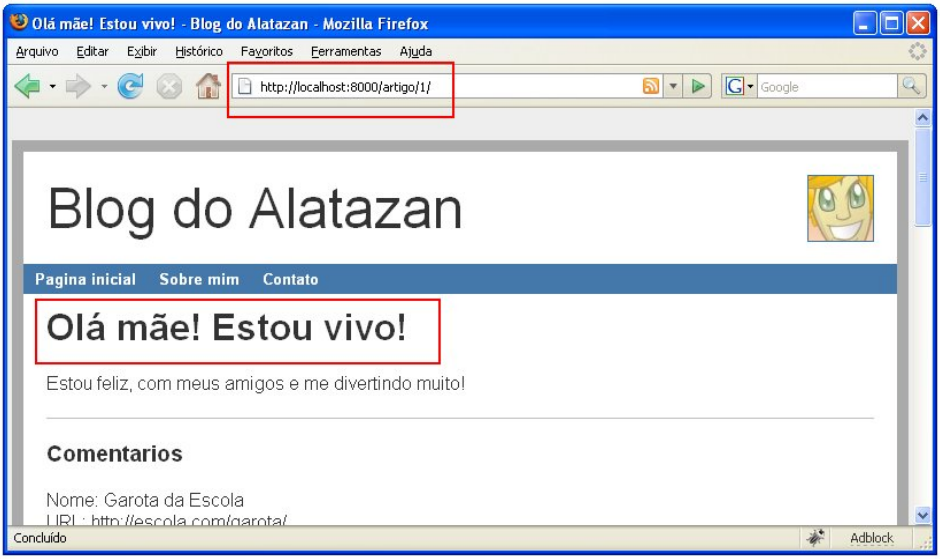
E sem dúvida ficaria mais fácil para pessoas **se identificarem com ela**. Mecanismos de **buscas** também ficam muito mais confortáveis com esse tipo de URL.

Agora, inicie o projeto em nosso ambiente de desenvolvimento, clicando duas vezes no arquivo **"executar.bat"** da pasta do projeto (**"C:\Projetos\meu_blog"**).

Abra o navegador e carregue a seguinte URL:

```
http://localhost:8000/artigo/1/
```

A seguinte página será carregada:



Um artigo com o título **"Olá mãe! Estou vivo!"** poderia muito bem ter uma URL como esta abaixo, não?

```
http://localhost:8000/artigo/ola-mae-estou-vivo/
```

Muito mais amigável e sem dúvida mais fácil de identificar. Então vamos lá!

Acrescentando um novo campo de slug à classe de modelo

Na pasta da aplicação **"blog"**, carregue o arquivo **"models.py"** para edição e localize a seguinte linha:

```
    publicacao = models.DateTimeField(
        default=datetime.now,
        blank=True
    )
```

Logo abaixo dela, acrescente a seguinte linha de código:

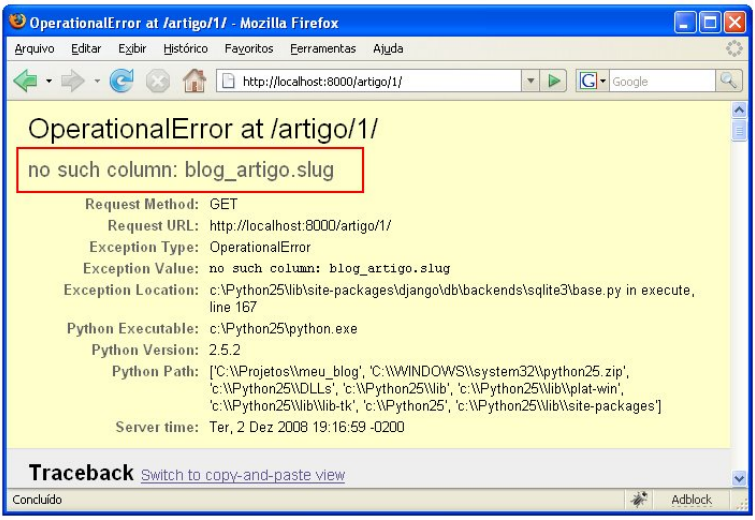
```
    slug = models.SlugField(max_length=100, blank=True)
```

Observe que definimos uma **largura máxima de 100 caracteres**, para que ele tenha a mesma largura máxima do campo **"titulo"**.

Observe também que definimos este campo com **blank=True**, o que significa que é um campo que **não precisa ser preenchido**.

Salve o arquivo. Feche o arquivo.

Agora, volte ao navegador e atualize pressionando a tecla **F5**. Veja o que acontece:



A mensagem de erro é enfática: **não existe a coluna 'blog_artigo.slug'**. Ou seja, não existe o campo **"slug"** na tabela **"blog_artigo"**.

Agora que a tabela **"blog_artigo"** já existe no banco de dados, com os próprios recursos do Django não é possível gerar automaticamente mudanças como a criação da nova coluna **"slug"**. Isso porque esse processo nunca é de fato tão simples quanto parece.

O Django possui um comando chamado **"dbshell"** para executar com o arquivo **"manage.py"** da pasta do projeto. Este comando acessa o *shell* do banco de dados para que você faça qualquer tarefa ali, seja de definição do modelo de dados, consultas ou tarefas de manutenção. Infelizmente esse comando não trabalha bem no Windows.

No Windows, é preciso usar um pequeno aplicativo de *shell* do **SQLite**. Portanto, vamos baixá-lo, certo?

Vá até a seguinte página e faça o download:

```
http://www.sqlite.org/download.html
```

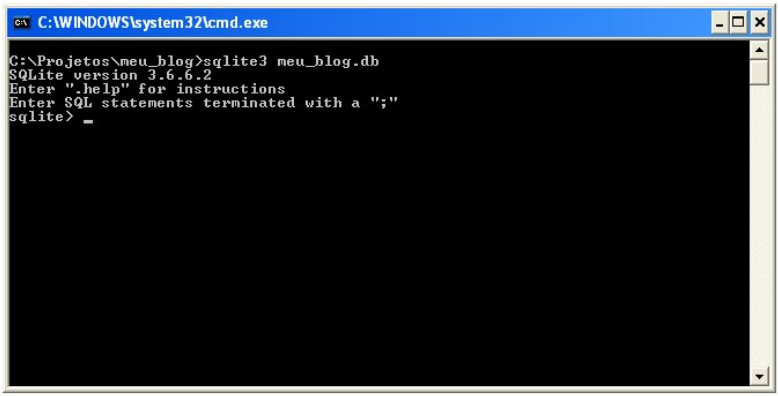
Localize o bloco que inicia com **"Precompiled Binaries For Windows"**. Faça o download do primeiro item da lista, que tem um nome semelhante a este:

```
sqlite-3_6_6_2.zip
```

Ao concluir o download, extraia o único arquivo (**"sqlite3.exe"**) para a pasta do projeto. Agora para usá-lo de uma maneira fácil, crie um novo arquivo chamado **"dbshell.bat"** na pasta do projeto com a seguinte linha de código:

```
sqlite3.exe meu_blog.db
```

Salve o arquivo. Feche o arquivo. Clique duas vezes sobre ele para executá-lo, e veja o resultado:

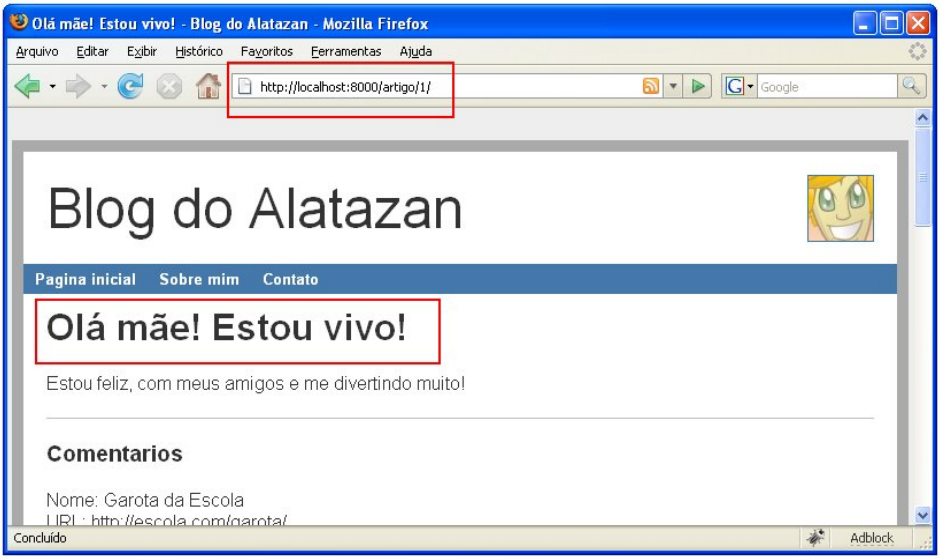


Estamos dentro do shell do **SQLite**.

Agora para criar a nova coluna, digite a seguinte expressão SQL no shell do SQLite:

```
alter table blog_artigo add slug varchar(200);
```

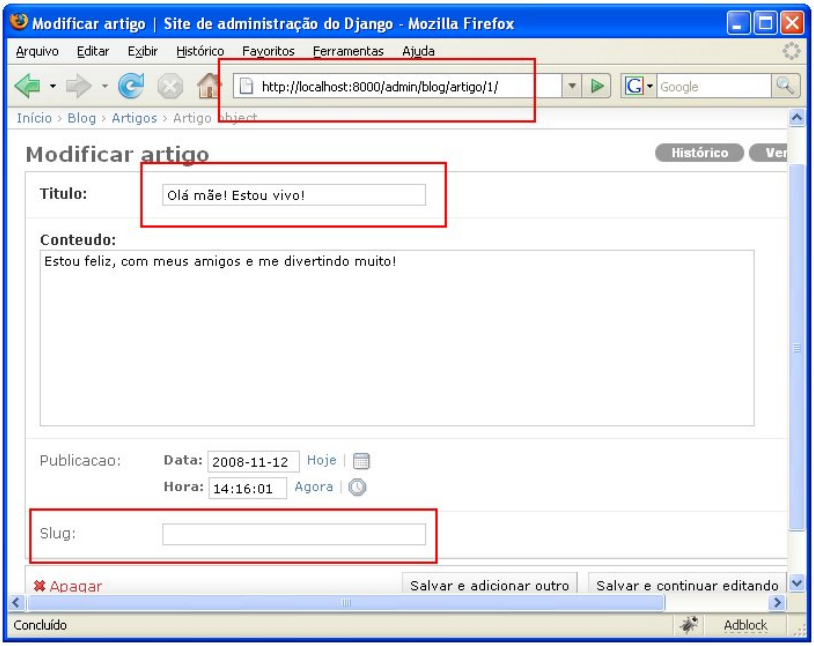
Feito isso, feche a janela do shell e volte ao navegador. Atualize a página com a tecla **F5** e veja que voltamos ao estado normal da página.



Agora vá à página de manutenção desse artigo no **Admin**, para ver como está o campo **"slug"**:

```
http://localhost:8000/admin/blog/artigo/1/
```

A apresentação da página é esta:



Sim, o campo está vazio. E não vá preenchê-lo por você próprio. Não faz sentido se o Django faz isso automaticamente. Mas como? Precisamos de definir um **signal**, um recurso do Django para agendar eventos.

Criando um signal

Os **Signals** no Django são como **triggers** em bancos de dados. E como eventos em programação gráfica. Basicamente a coisa funciona assim:

1. O artigo é modificado;
2. Quando esse artigo é salvo, antes que as mudanças sejam enviadas para o banco de dados, os signals do tipo **"pre_save"** da classe de modelo **Artigo** são executados, um a um, em sequência;
3. Caso nada ocorra de errado as modificações são gravadas no banco de dados;
4. Depois disso, os signals do tipo **"post_save"** da mesma classe de modelo são executados, também em sequência.

Portanto, vamos criar um signal de **"pre_save"** para preencher o campo **"slug"** automaticamente antes que os dados sejam gravados no banco de dados.

Para isso, vá até a pasta da aplicação **"blog"**, abra o arquivo **"models.py"** para

edição e acrescente as seguintes linhas ao final do arquivo:

```
# SIGNALS
from django.db.models import signals
from django.template.defaultfilters import slugify


def artigo_pre_save(signal, instance, sender, **kwargs):
    instance.slug = slugify(instance.titulo)


signals.pre_save.connect(artigo_pre_save, sender=Artigo)
```

Resumindo o que fizemos:

Nós importamos os pacotes de **signals** para classes de modelo e a função **"slugify"**, que transforma qualquer *string* para formato de **slug**:

```
# SIGNALS
from django.db.models import signals
from django.template.defaultfilters import slugify
```

Definimos a função a ser conectada ao **signal**. Esta função será executada toda vez que um **artigo** for salvo, antes que as mudanças sejam persistidas no banco de dados:

```
def artigo_pre_save(signal, instance, sender, **kwargs):
```

A linha seguinte faz exatamente o que precisamos: atribui ao campo **"slug"** do **artigo** em questão o valor do campo **"titulo"**, formatado para slug:

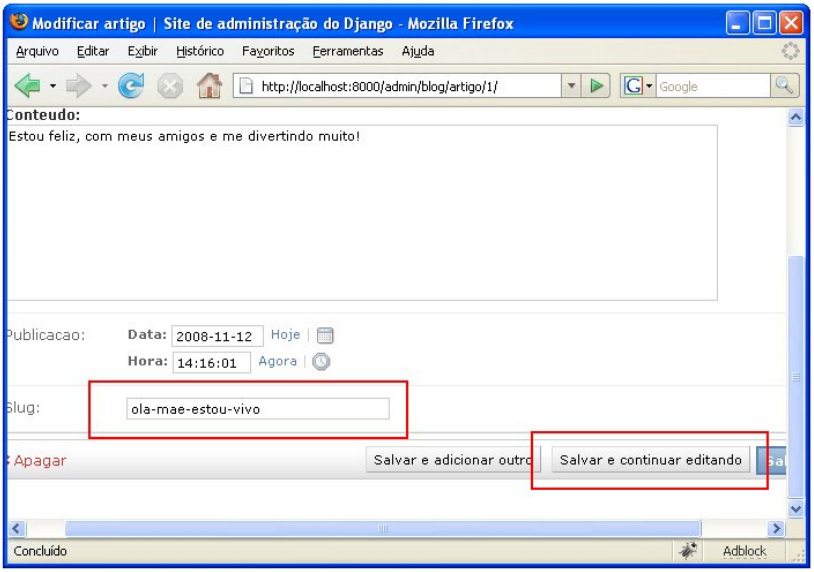
```
    instance.slug = slugify(instance.titulo)
```

Por fim, conectamos a função ao *signal*, dizendo para o Django que a função deve ser executada pelo signal **"pre_save"**, quando este for enviado pela classe **"Artigo"**:

```
signals.pre_save.connect(artigo_pre_save, sender=Artigo)
```

Salve o arquivo. Feche o arquivo.

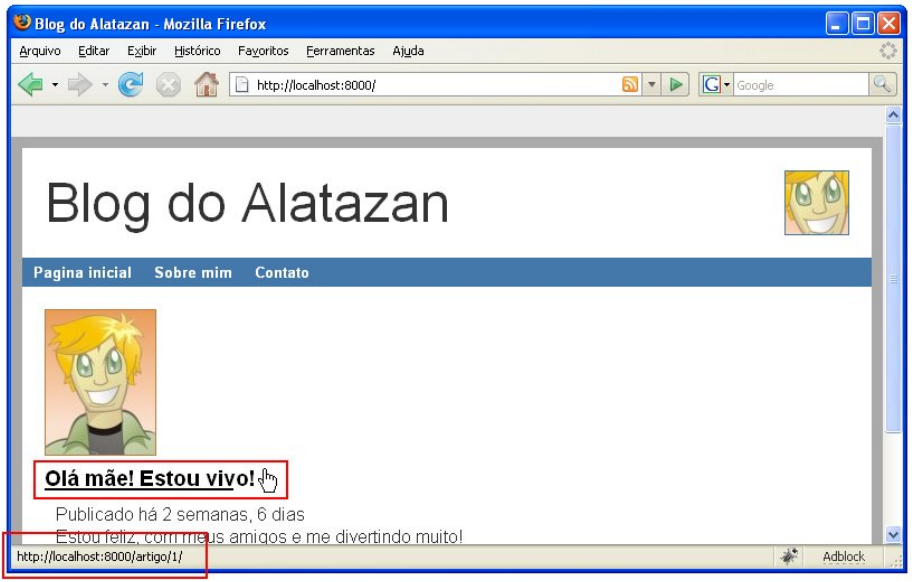
Volte ao navegador, na página do artigo no **Admin** e apenas clique sobre o botão **"Salvar e continuar editando"** para ver o efeito do que fizemos, veja com seus próprios olhos:



Nosso signal está funcionando bacana, certo? Agora carregue a página principal do blog:

```
http://localhost:8000/
```

Mova o mouse sobre o **título do artigo** (que é um **link**) e veja que a URL do artigo permanece da mesma forma:



Isso é porquê, de fato, não fizemos nada nesse sentido. Agora vá à pasta **"blog/templates/blog/"**, abra o arquivo **"artigo_archive.html"** para edição e localize a seguinte linha:

```
<a href="{% url blog.views.artigo artigo_id=artigo.id %}">
    <h2>{{ artigo.titulo }}</h2>
</a>
```

Modifique esta linha para ficar assim:

```
<a href="{{ artigo.get_absolute_url }}">
    h2>{{ artigo.titulo }}</h2>
</a>
```

Salve o arquivo. Feche o arquivo.

Porque estamos fazendo isso?

Quando criamos a listagem do blog, no **capítulo 4 - "Criando um Blog maneiro"** - informamos a template tag **"{% url %}"** para obter a URL do artigo e criar nosso link com ela. Depois, quando habilitamos o recurso de RSS no **capítulo 6 - "O RSS é o entregador fiel"**, definimos o método **"get_absolute_url()"** na classe de modelo **Artigo**, também para representar a mesma URL. Ou seja, para dois casos diferentes, usamos duas formas também diferentes de obter a **mesma URL**: a URL do artigo.

Isso não estava certo.

Portanto agora estamos fazendo um casamento das duas idéias: acabamos de ajustar o template da lista de artigos para usar o método **"get_absolute_url()"**, e vamos agora ajustar esse método para trabalhar da mesma forma que trabalha a template tag **"{% url %}"**.

Vá até à pasta da aplicação **"blog"** e abra o arquivo **"models.py"** para edição. Localize o seguinte trecho de código:

```
def get_absolute_url(self):
     return '/artigo/%d/'%self.id
```

Modifique para ficar assim:

```
def get_absolute_url(self):
     return reverse(
     'blog.views.artigo',
     kwargs={'slug': self.slug}
     )
```

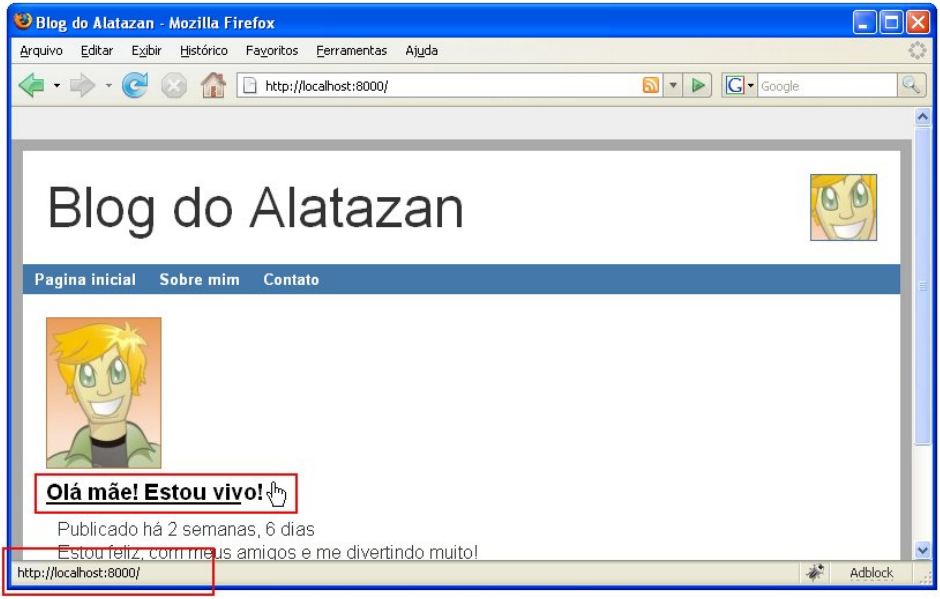
E como a função **"reverse"** é um elemento estranho aqui, vamos importá-la no início do arquivo. Localize a seguinte linha:

```
from django.db import models
```

E acrescente logo abaixo dela:

```
from django.core.urlresolvers import reverse
```

Salve o arquivo. Feche o arquivo. Agora volte ao navegador, atualize a página com **F5** e veja o que acontece:



Mas o que aconteceu? Perdemos o link? Sim. Perdemos porque ainda falta uma coisa.

Na definição da URL do artigo há uma referência a seu campo **"id"**, mas nós acabamos de definir a seguinte linha de código, que agora faz uso do **"slug"** em seu lugar:

```
        return reverse(
       'blog.views.artigo',
        kwargs={'slug': self.slug}
        )
```

Portanto, na pasta do projeto, abra o arquivo **"urls.py"** para edição e localize a seguinte linha de código:

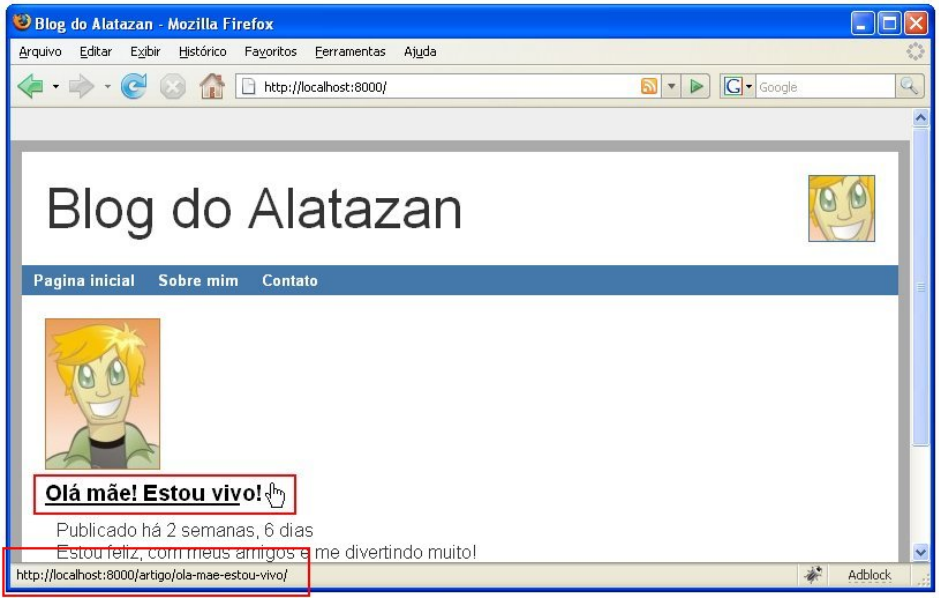
```
    (r'^artigo/(?P<artigo_id>\d+)/$', 'blog.views.artigo'),
```

Modifique para ficar assim:

```
    (r'^artigo/(?P<slug>[\w_-]+)/$', 'blog.views.artigo'),
```

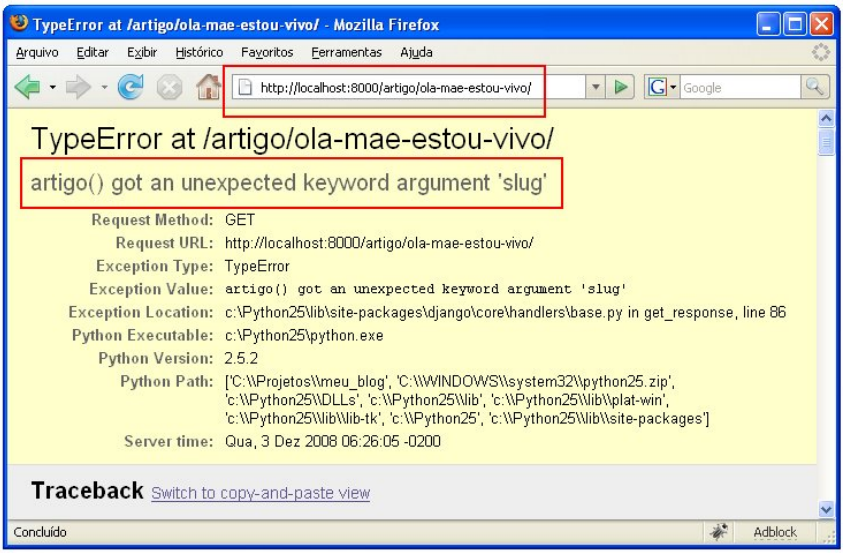
Você notou que trocamos o **"artigo_id"** pelo **"slug"**? Você vai descobrir também que essa expressão regular exige **letras, números e os caracteres "_" e "-"**, quando define **"[\w_-]+"** no lugar de **"\d+"**

Salve o arquivo. Feche o arquivo. Volte ao navegador, atualize a página e veja como ficou:



Até que enfim, temos uma URL amigável!

Agora clique sobre o link, e... mais uma tela de erro!



Esse erro ocorreu porque precisamos também ajustar a **view** do artigo, para aceitar o parâmetro **"slug"**.

Então, na pasta da aplicação **"blog"**, abra o arquivo **"views.py"** para edição e localize o seguinte trecho de código:

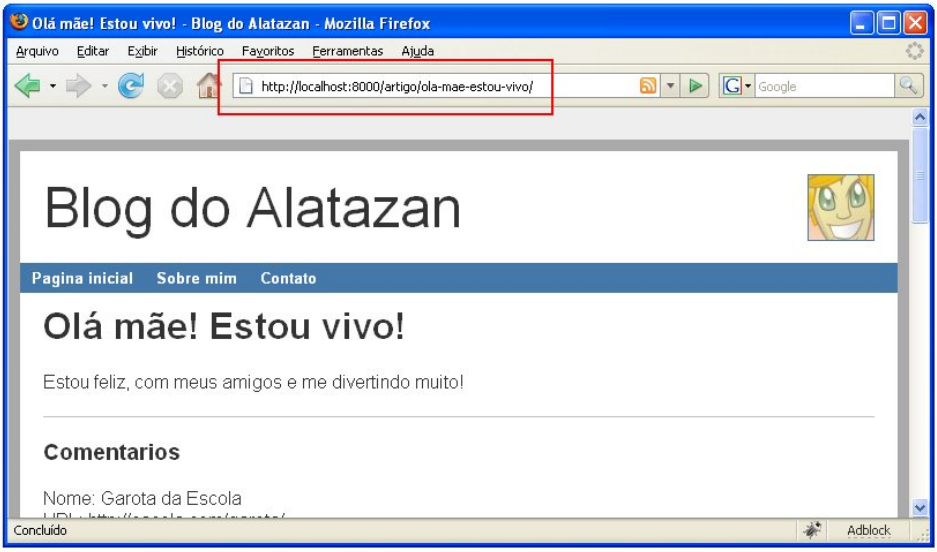
```
def artigo(request, artigo_id):
    artigo = Artigo.objects.get(id=artigo_id)
    return render_to_response('blog/artigo.html', locals(),
        context_instance=RequestContext(request))
```

Agora modifique para ficar assim:

```
def artigo(request, slug):
    artigo = Artigo.objects.get(slug=slug)
    return render_to_response('blog/artigo.html', locals(),
        context_instance=RequestContext(request))
```

Notou que trocamos as referências ao campo **"id"** para o campo **"slug"** do artigo?

Salve o arquivo. Volte ao navegador, atualize a página com **F5** e veja o resultado:



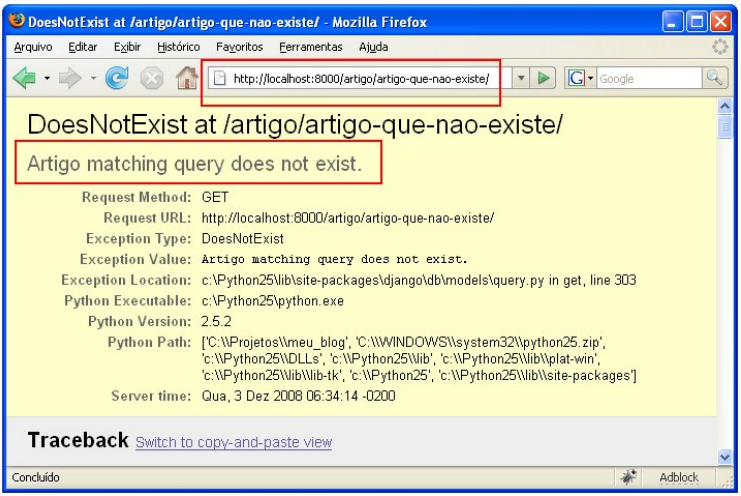
Satisfeito? Sim, claro, mas vamos tentar uma coisa diferente agora...

Usando get_object_or_404

.. carregue a seguinte URL no navegador:

```
http://localhost:8000/artigo/artigo-que-nao-existe/
```

E veja o que ocorre quando se tenta carregar a página de um artigo que não existe:



A mensagem diz o que já dissemos:

```
Artigo matching query does not exist.
```

Ou seja: **este artigo não existe**. Mas nós podemos melhorar isso. Volte ao arquivo **"views.py"** da pasta da aplicação **"blog"** e localize esta linha:

```
    artigo = Artigo.objects.get(slug=slug)
```

Agora modifique, para ficar assim:

```
    artigo = get_object_or_404(Artigo, slug=slug)
```

Em outras palavras: a partir de agora, quando o artigo com o slug informado não for encontrado, será retornado um erro do tipo **"Página não encontrada"**, o famoso **erro 404**.

Precisamos também importar a função **"get_object_or_404()"**. Para isso localize esta outra linha:

```
from django.template import RequestContext
```

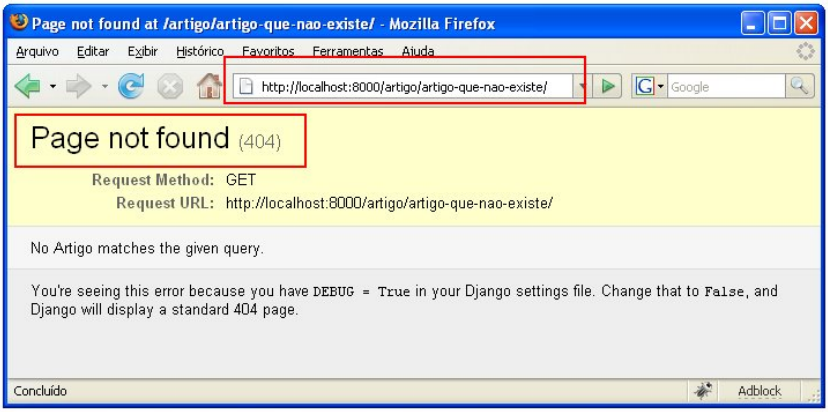
E acrescente esta abaixo dela:

```
from django.shortcuts import get_object_or_404
```

Agora, o arquivo **"views.py"** está assim:

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from django.shortcuts import get_object_or_404


from models import Artigo


def artigo(request, slug):
    artigo = get_object_or_404(Artigo, slug=slug)
    return render_to_response('blog/artigo.html', locals(),
        context_instance=RequestContext(request))
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, atualize com **F5** e veja como ficou:



Pronto! Mais uma questão resolvida!

Evitando duplicidades

Vamos agora testar outra coisa importante. Vá à seguinte URL para criar um novo artigo:

```
http://localhost:8000/admin/blog/artigo/add/
```

Preencha os campos assim:

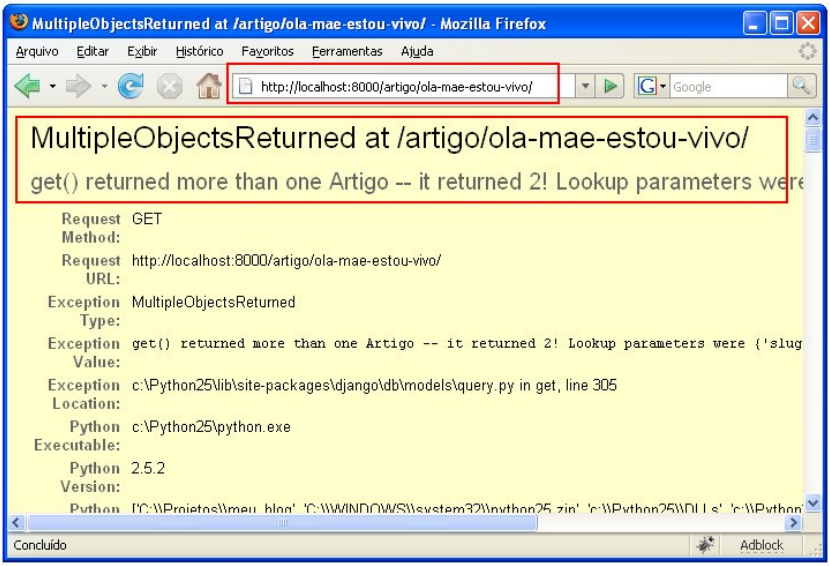
- Título: **"Olá mãe! Estou vivo!"**
- Conteúdo: **"Outro artigo com o mesmo título só pra ver no que dá"**

Clique sobre o botão **"Salvar"**.

Agora tente carregar a URL do artigo:

```
http://localhost:8000/artigo/ola-mae-estou-vivo/
```

Veja o que aparece:



A mensagem é bastante clara:

```
MultipleObjectsReturned at /artigo/ola-mae-estou-vivo/

get() returned more than one Artigo -- it returned 2! Lookup
parameters were {'slug': u'ola-mae-estou-vivo'}
```

Resumindo: existem dois artigos com o **mesmo slug**, e o Django não sabe qual dos dois trazer! Só há uma forma de resolver isso: garantindo que não vão existir dois artigos com o mesmo slug.

Então, na pasta da aplicação **"blog"**, abra o arquivo **"models.py"** para edição e localize a seguinte linha:

```
slug = models.SlugField(max_length=100, blank=True)
```

Modifique para ficar assim:

```
slug = models.SlugField(max_length=100, blank=True, unique=True)
```

Com **"unique=True"**, o campo passa a ser tratado no banco de dados como **"índice único"**, para garantir que de que não haverão dois registros com um mesmo valor repetido nesse campo.

Mas só isso não é suficiente. Localize este trecho de código:

```
def artigo_pre_save(signal, instance, sender, **kwargs):

    instance.slug = slugify(instance.titulo)
```

Ele está muito vulnerável. Podemos melhorar isso para ajustar o slug caso ele já exista, permitindo que seu **título** seja repetido sem que seu **slug** o seja. Portanto,

modifique esta função de signal para ficar assim:

```
def artigo_pre_save(signal, instance, sender, **kwargs):

    """Este signal gera um slug automaticamente. Ele verifica se
ja existe um artigo com o mesmo slug e acrescenta um numero ao
final para evitar duplicidade"""

    if not instance.slug:

        slug = slugify(instance.titulo)

        novo_slug = slug

        contador = 0


        while Artigo.objects.filter(

        slug=novo_slug

        ).exclude(id=instance.id).count() > 0:

            contador += 1

            novo_slug = '%s-%d'%(slug, contador)


        instance.slug = novo_slug
```

Salve o arquivo. Vamos detalhar cada parte do que fizemos?

Aqui nós definimos que vamos gerar o slug somente se o campo estiver vazio:

```
    if not instance.slug:
```

Geramos o **slug** do **título** do artigo e o armazenamos em duas variáveis: **"slug"** e **"novo_slug"**. Para quê isso? É porque logo à frente, podemos modificar o valor de **"novo_slug"**, mantendo o valor original da variavel **"slug"** como referência:

```
        slug = slugify(instance.titulo)

        novo_slug = slug
```

Iniciamos uma variável **"contador"** com valor zero. Ela irá somar à variável **"slug"** em caso de haver duplicidade para acrescentar um número adicional ao final do slug:

```
        contador = 0
```

Aqui, teremos um **laço**, **enquanto** a quantidade de **Artigos** com o slug contido na variável **"novo_slug"** for **maior que zero**, será incrementado o contador e atribuído à variável **"novo_slug"**, para que a tentativa seja feita novamente:

```
        while Artigo.objects.filter(

        slug=novo_slug

        ).exclude(id=instance.id).count() > 0:
```

```
            contador += 1
            novo_slug = '%s-%d'%(slug, contador)
```

Por fim, depois que sair do laço **"while"**, já com a variável **"novo_slug"** com o valor ideal, este valor é atribuído ao campo **"slug"** do objeto:

```
        instance.slug = novo_slug
```

Vamos tomar como exemplo o caso do artigo com título **"Olá mãe! Estou vivo!"**:

1. O slug gerado será "ola-mae-estou-vivo";
    1. Se não forem encontrados outros artigos com o slug gerado, então ok;
2. Caso o contrário, acrescenta um número "1" ao final, e o novo slug passa a ser "ola-mae-estou-vivo-1";
    1. Se não forem encontrados outros artigos com o novo slug, então ok;
3. Caso o contrário, acrescenta um número "2" ao final, e o novo slug passa a ser "ola-mae-estou-vivo-2";
    1. Se não forem encontrados outros artigos com o novo slug, então ok;
4. Caso o contrário, acrescenta um número "3" ao final, e o novo slug passa a ser "ola-mae-estou-vivo-3";
5. E assim por diante...

No fim da história, o arquivo **"models.py"** todo ficou assim:

```
from datetime import datetime
from django.db import models
from django.core.urlresolvers import reverse


class Artigo(models.Model):
    class Meta:
        ordering = ('-publicacao',)


    titulo = models.CharField(max_length=100)
    conteudo = models.TextField()
    publicacao = models.DateTimeField(
    default=datetime.now, blank=True
```

```
    )
    slug = models.SlugField(
    max_length=100,
    blank=True,
    unique=True
    )

    def get_absolute_url(self):
        return reverse(
        'blog.views.artigo',
        kwargs={'slug': self.slug}
        )

# SIGNALS
from django.db.models import signals
from django.template.defaultfilters import slugify

def artigo_pre_save(signal, instance, sender, **kwargs):
    """Este signal gera um slug automaticamente. Ele verifica se
ja existe um artigo com o mesmo slug e acrescenta um numero ao
final para evitar duplicidade"""
    if not instance.slug:
        slug = slugify(instance.titulo)
        novo_slug = slug
        contador = 0

        while Artigo.objects.filter(
        slug=novo_slug
        ).exclude(id=instance.id).count() > 0:
            contador += 1
            novo_slug = '%s-%d'%(slug, contador)

        instance.slug = novo_slug

signals.pre_save.connect(artigo_pre_save, sender=Artigo)
```

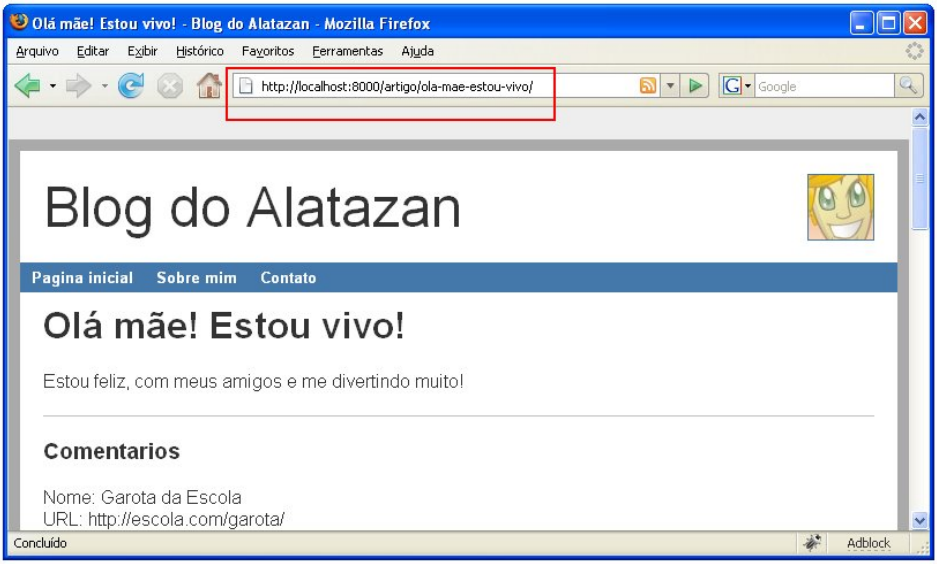
Feche o arquivo. Volte ao navegador na página do novo artigo no **Admin**:

```
http://localhost:8000/admin/blog/artigo/2/
```

Remova todo o conteúdo do campo **"slug"**, deixando-o **vazio** e clique sobre o botão **"Salvar"**. Agora volte à URL do artigo:

```
http://localhost:8000/artigo/ola-mae-estou-vivo/
```

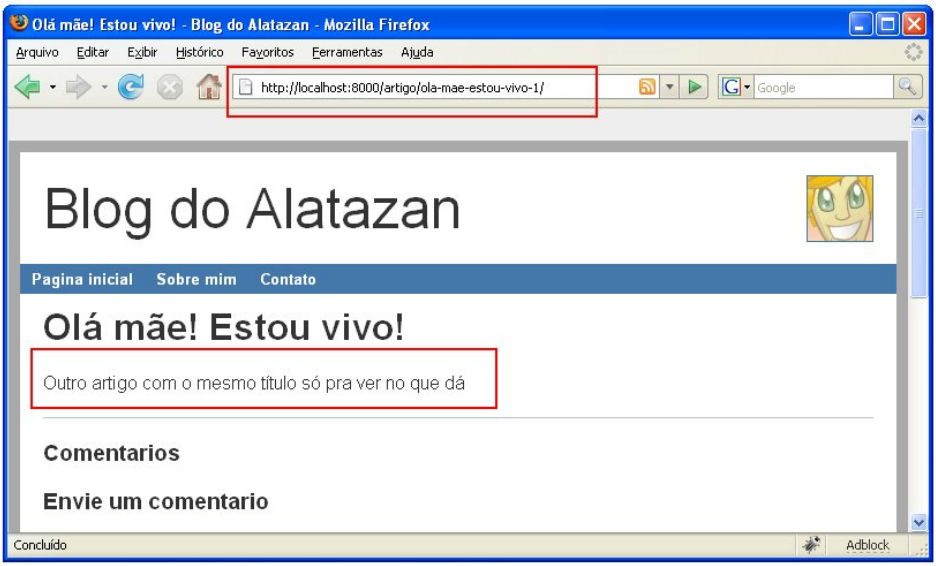
Veja que agora dá tudo certo:



Tudo certo. E carregando esta outra URL:

```
http://localhost:8000/artigo/ola-mae-estou-vivo-1/
```

Veja que tudo dá certo também:



Então, e agora? Que tal um álbum de fotos?

Alatazan sente cada vez mais consistência no que aprende, mas uma coisa está muito clara para ele:

- Tudo isso é muito bacana, mas a cada dia que passa, cada página virada, percebo que há muito mais a aprender...
- Alatazan, isso é porque o que vemos aqui é apenas um exemplo dos vários recursos que o Django oferece para cada tema.
- Cartola entrou complementando Nena:
- Sim, e falando de **signals**, por exemplo: existem vários deles. Hoje só usamos o **"pre_save"** para classes de modelo, e citamos o **"post_save"**, mas existem signals também para **exclusão**, para inicialização da classe e diversos outros. Você não pode ficar só aqui, no que falamos, há um universo todo ao nosso redor...

A vida é realmente irônica... Cartola pouco sabe que Alatazan conhece muito mais sobre a imensidão do universo do que qualquer terráqueo, mas...

- Tá, então podemos dizer que o resumo do que aprendemos hoje é este:
  - Slugs são identificadores para evitar endereços complexos e ajudar os usuários e mecanismos de busca a encontrarem nossas páginas;

• Para criar um novo campo no banco de dados é preciso saber um pouquinho de SQL;

• Ou ter uma ferramenta gráfica para fazer isso... - interrompeu Nena.

Alatazan continuou:

• Certo... uma URL de slug deve ter a expressão **[\w_-]+** para garantir que serão informados apenas slugs válidos;

• Signals devem ser definidos no arquivo **"models.py"**. São funções, com argumentos específicos, que são **conectadas** aos signals. Elas serão executadas uma por uma quando aquele **evento** acontece;

• Usar o método **get_absolute_url()** evita repetições;

• Usar a função **reverse()** dentro do método **get_absolute_url()** é a forma ideal para evitar mais repetições;

• Usar **get_object_or_404()** ajuda muito a deixar as coisas mais claras para o usuário;

• Quando a assinatura de uma **URL** é modificada, a **view** também deve ser ajustada a isso;

• Para garantir que não haverão duplicações, devo usar **unique=True** e ajustar o signal com um código consistente e claro...

• Ufa! É isso!

Cartola reagiu, empolgado:

• Aí, meu chapa, falou bonito hein! Está pegando o jeito no português!

E aí Nena tratou de encerrar o dia...

• Pois falando em bonito, amanhã vamos fazer uma nova aplicação, uma galeria de fotos para o seu blog!

Capítulo 20: Uma galeria de imagens simples e útil

Uma das coisas que Alatazan mais gostou no Planeta Terra foram as **pinturas** de alguns artistas plásticos, especialmente os **impressionistas** e os **cubistas**.

A princípio ele achava que "impressionistas" eram aqueles caras corpulentos e de topete duro, queixo comprido e olhar de galã, vestindo uma tanguinha vermelha com uma pequena flor ao lado esquerdo.

Não, não... umas semanas depois ele descobriu que isso se tratava de outro movimento. Impressionistas eram quase fotógrafos, só que mais generosos um pouco e com a diferença de precisarem de um oftalmologista urgente. Que vida dura a daquela época...

Então uma das coisas que Alatazan mais queria era uma **galeria de imagens** em seu site. Assim poderia mostrar as artes daqui para a sua mãe, numa tentativa a mais de fazê-la se interessar por arte.

Uma nova aplicação no projeto

A primeira coisa a observar aqui é que a área de domínio de uma "galeria de imagens" está **fora do foco** de um blog. É possível ter um blog ou uma galeria de imagens, ou simplesmente **ter os dois**. Eles não são **dependentes** entre si, nem mesmo **excludentes**.

Por isso, começamos aqui a conhecer algumas das prática para se construir **aplicações plugáveis**.

Criando a nova aplicação

Na pasta do projeto, crie uma nova pasta, chamada **"galeria"**. Dentro desta pasta crie um novo arquivo vazio, chamado **"__init__.py"**.

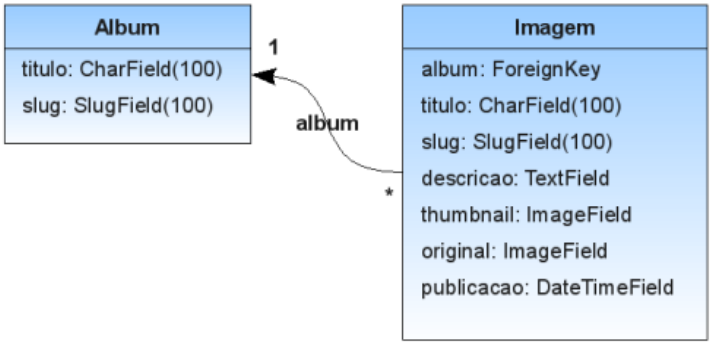
Agora, na mesma pasta, crie um arquivo chamado **"models.py"** e escreva o seguinte código dentro:

```
from datetime import datetime
from django.db import models


class Album(models.Model):
    """Um album eh um pacote de imagens, ele tem um titulo e um
slug para sua identificacao."""
    class Meta:
        ordering = ('titulo',)


    titulo = models.CharField(max_length=100)
    slug = models.SlugField(
    max_length=100,
    blank=True,
    unique=True
    )


    def __unicode__(self):
        return self.titulo


class Imagem(models.Model):
    """Cada instancia desta classe contem uma imagem da galeria,
com seu respectivo thumbnail (miniatura) e imagem em tamanho
natural. Varias imagens podem conter dentro de um Album"""


    class Meta:
        ordering = ('album','titulo',)


    album = models.ForeignKey('Album')
    titulo = models.CharField(max_length=100)
    slug = models.SlugField(
    max_length=100,
    blank=True,
    unique=True
    )
    descricao = models.TextField(blank=True)
```

```
    original = models.ImageField(
        null=True,
        blank=True,
        upload_to='galeria/original',
        )
    thumbnail = models.ImageField(
        null=True,
        blank=True,
        upload_to='galeria/thumbnail',
        )
    publicacao = models.DateTimeField(
    default=datetime.now,
    blank=True
    )

    def __unicode__(self):
        return self.titulo
```

Puxa! Que tanto de coisas de uma só vez!

Bom, a representação em diagrama dessas duas classes de modelo é esta:



Vamos falar do que é novidade...

O trecho abaixo determina que a classe de modelo possui uma ordenação padrão pelos campos **"album"** e **"titulo"**, em ordem ascendente:

```
    class Meta:
```

```
        ordering = ('album','titulo',)
```

Já a linha seguinte, aponta uma **"Foreign Key"**, ou seja, um relacionamento da classe **"Imagem"** para a classe **"Album"**, o que significa que **uma imagem faz parte de um álbum**, e um álbum pode conter **muitas** delas:

```
    album = models.ForeignKey('Album')
```

Por fim, este outro trecho abaixo declara os campos para armazenamento de imagem. O primeiro campo (**"original"**) deve armazenar a imagem exatamente como esta foi enviada. Já o campo seguinte (**"thumbnail"**) deve armazenar sua miniatura, gerada automaticamente pela nossa aplicação.

```
    original = models.ImageField(
        null=True,
        blank=True,
        upload_to='galeria/original',
        )
    thumbnail = models.ImageField(
        null=True,
        blank=True,
        upload_to='galeria/thumbnail',
        )
```

Outra coisa importante nos campos declarados acima é o argumento **"upload_to"** presente em cada um deles. Esse argumento define que as imagens enviadas para o campo em questão devem ser armazenadas na pasta indicada pela setting **"MEDIA_ROOT"**, somada ao caminho atribuído ao argumento.

O campo **"original"** por exemplo indica que o caminho completo, partindo da pasta do projeto deve ser:

```
media/galeria/original/
```

E o campo **"thumbnail"** indica este outro caminho:

```
media/galeria/thumbnail/
```

Salve o arquivo. Feche o arquivo.

Criando uma única função de signal para várias classes

Mas há ainda outra coisa importante para tratar. Você notou que ambas das classes possuem um campo **"slug"** seguindo a mesma linha de pensamento que tratamos no capítulo anterior? Pois é... isso significa que ambas precisam daquele **signal** que criamos para gerar o **slug** a partir do campo **"titulo"**.

Mas para evitar a repetição de código, vamos fazer agora uma coisa diferente: na pasta da aplicação **"blog"**, abra o arquivo **"models.py"** para edição e localize este trecho de código:

```
# SIGNALS
from django.db.models import signals
from django.template.defaultfilters import slugify


def artigo_pre_save(signal, instance, sender, **kwargs):
    """Este signal gera um slug automaticamente. Ele verifica se
ja existe um artigo com o mesmo slug e acrescenta um numero ao
final para evitar duplicidade"""
    if not instance.slug:
        slug = slugify(instance.titulo)
        novo_slug = slug
        contador = 0

        while Artigo.objects.filter(
        slug=novo_slug
        ).exclude(id=instance.id).count() > 0:
            contador += 1
            novo_slug = '%s-%d'%(slug, contador)

        instance.slug = novo_slug


signals.pre_save.connect(artigo_pre_save, sender=Artigo)
```

Modique para ficar assim:

```
# SIGNALS
from django.db.models import signals
from utils.signals_comuns import slug_pre_save


signals.pre_save.connect(slug_pre_save, sender=Artigo)
```

Levou um susto? Mas é isso mesmo! Nós importamos a função **"slug_pre_save"** do módulo **"utils.signals_comuns"** que vai servir ao signal da classe **"Artigo"**.

Salve o arquivo. Feche o arquivo.

Agora voltando à pasta da aplicação **"galeria"**, abra novamente o arquivo **"models.py"** para edição e acrescente estas linhas ao final:

```
# SIGNALS
from django.db.models import signals
from utils.signals_comuns import slug_pre_save


signals.pre_save.connect(slug_pre_save, sender=Album)
signals.pre_save.connect(slug_pre_save, sender=Imagem)
```

Observe que fizemos a mesma coisa que fizemos com a classe **"Artigo"** da aplicação **"blog"**.

Salve o arquivo. Feche o arquivo.

Agora na pasta do projeto, crie uma pasta chamada **"utils"** e dentro dela o arquivo **"__init__.py"** vazio. Crie também o arquivo **"signals_comuns.py"** com o seguinte código dentro:

```
from django.template.defaultfilters import slugify


def slug_pre_save(signal, instance, sender, **kwargs):
    """Este signal gera um slug automaticamente. Ele verifica se
ja existe um objeto com o mesmo slug e acrescenta um numero ao
final para evitar duplicidade"""
    if not instance.slug:
        slug = slugify(instance.titulo)
        novo_slug = slug
        contador = 0


        while sender.objects.filter(
        slug=novo_slug
        ).exclude(id=instance.id).count() > 0:
            contador += 1
            novo_slug = '%s-%d'%(slug, contador)


        instance.slug = novo_slug
```

Observe que há ali uma sutil diferença: nós trocamos a palavra **"Artigo"** para **"sender"**:

```
        while sender.objects.filter(
```

```
        slug=novo_slug
        ).exclude(id=instance.id).count() > 0:
```

A variável **"sender"** dentro da função de signal representa a **classe** que enviou o signal. A variável **"instance"** representa o objeto em questão. E desde que os campos relacionados ao assunto sejam **"titulo"** e **"slug"**, essa função vai trabalhar bem em todos os casos.

Salve o arquivo. Feche o arquivo.

Instalando a nova aplicação no projeto

Agora precisamos instalar a nova aplicação **"galeria"** no projeto. Para isso, abra o arquivo **"settings.py"** da pasta do projeto para edição e localize a setting **"INSTALLED_APPS"** e ao final da tupla (logo abaixo da linha da aplicação **"blog"**), acrescente a seguinte linha:

```
    'galeria',
```

Para ficar assim:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.syndication',
    'django.contrib.flatpages',
    'django.contrib.comments',


    'blog',
    'galeria',
)
```

Salve o arquivo. Feche o arquivo.

Agora vamos gerar as novas tabelas no banco de dados. Na pasta do projeto, clique duas vezes sobre o arquivo **"gerar_banco_de_dados.bat"** e veja que aparece o seguinte erro na tela do **MS-DOS**:

```
Error: One or more models did not validate:
galeria.imagem: "original": To use ImageFields, you need to
install the Python Imaging Library.
```

```
Get it at http://www.pythonware.com/products/pil/ .

galeria.imagem: "thumbnail": To use ImageFields, you need to
install the Python Imaging Library.

Get it at http://www.pythonware.com/products/pil/ .
```

A mesma mensagem é mostrada duas vezes: uma para cada campo.

Para os campos **"original"** e **"thumbnail"**, que são do tipo **"ImageField"** é necessária a instalação de uma biblioteca adicional, chamada **"Python Imaging Library"**.

Para fazer o download da **PIL** (o nome curto e mais conhecido para a Python Imaging Library), vá a esta URL:

```
http://www.pythonware.com/products/pil/
```

Na seção da página que inicia com **"Downloads"**, localize a versão mais adequada à versão do **Python** que você tem instalado. Nós trabalhamos no livro com a versão **2.5**, portanto, neste caso você deve clicar na seguinte opção:

```
Python Imaging Library 1.1.6 for Python 2.5 (Windows only)
```

Depois de concluído o download, execute o arquivo e clique sobre o botão **"Avançar"** até finalizar a instalação.

Agora, com a **PIL** instalada, clique novamente duas vezes no arquivo **"gerar_banco_de_dados.bat"** para gerar as novas tabelas no banco de dados. O resultado deve ser este:

```
Creating table galeria_album

Creating table galeria_imagem

Installing index for galeria.Imagem model
```

Ou seja, foram criadas duas novas tabelas para a aplicação **"galeria"**.

Configurando o admin para a nova aplicação

Agora, voltando à pasta da aplicação **"galeria"**, crie um novo arquivo, chamado **"admin.py"**, com o seguinte código dentro:

```
from django.contrib import admin
from django.contrib.admin.options import ModelAdmin


from models import Album, Imagem


class AdminAlbum(ModelAdmin):
    list_display = ('titulo',)
```

```
    search_fields = ('titulo',)


class AdminImagem(ModelAdmin):
    list_display = ('album','titulo',)
    list_filter = ('album',)
    search_fields = ('titulo','descricao',)


admin.site.register(Album, AdminAlbum)
admin.site.register(Imagem, AdminImagem)
```

Como pode ver, nosso arquivo de **"admin"** para a aplicação **"galeria"** está melhor elaborado do que é o da aplicação **"blog"**.

A classe **"ModelAdmin"** existe para se trabalhar melhor o **Admin** de uma classe de modelo. É possível por exemplo informar quais campos você quer mostrar na listagem (**"list_display"**) ou por quais campos você deseja filtrar os objetos (**"list_filter"**). Há ainda como informar por quais campos você deseja fazer buscas (**"search_fields"**) e muitos outros recursos que ainda vamos estudar um pouco melhor num capítulo futuro.

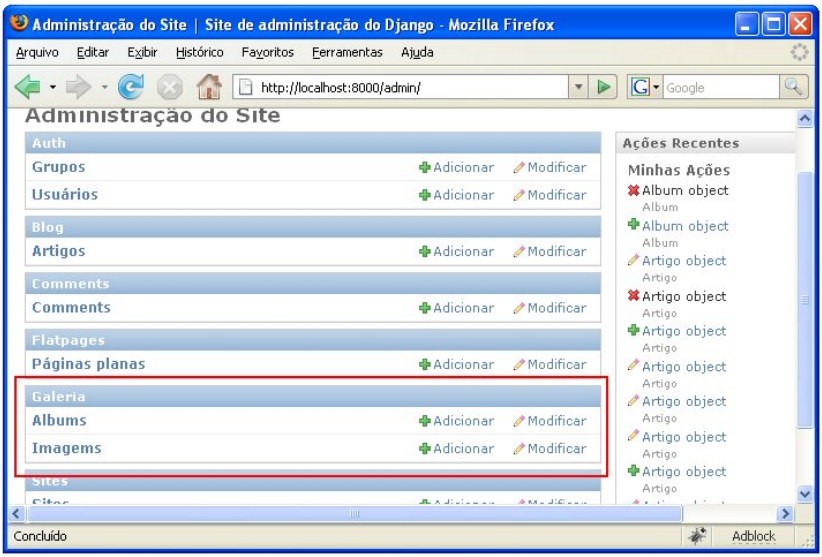
Salve o arquivo. Feche o arquivo.

Agora execute o seu projeto, clicando duas vezes sobre o arquivo **"executar.bat"** da pasta do projeto. Caso ele já estava em execução, verifique se ele não foi afetado pela **mensagem de erro** que pedia a instalação da **PIL**. Neste caso, apenas feche a janela e execute o arquivo novamente.

Agora vá ao navegador e carregue a URL do Admin:

```
http://localhost:8000/admin/
```

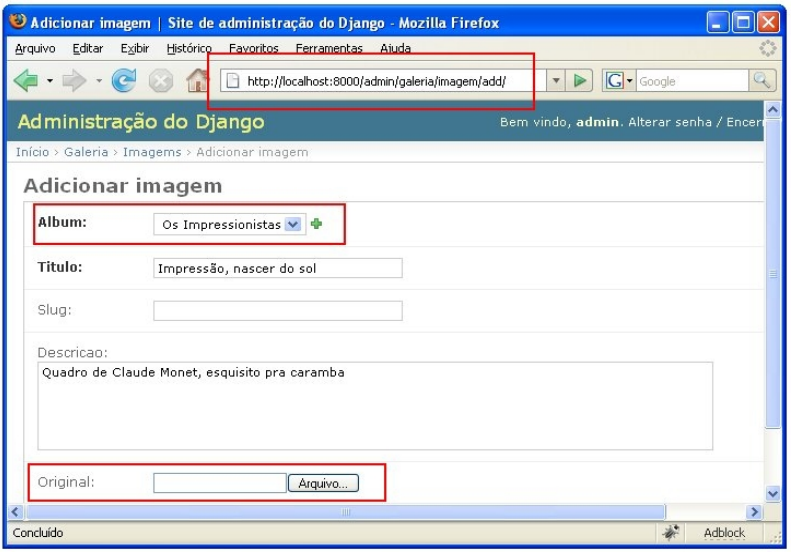
E a nova aplicação já mostra sua cara!



No entanto, para enviar uma nova imagem, antes é necessário criar as novas pastas, para onde os arquivos das imagens serão enviados.

Para isso, vá até a pasta **"media"** da pasta do projeto, crie uma nova pasta chamada **"galeria"**, e dentro dela crie outras duas: **"original"** e **"thumbnail"**.

Agora sim, você pode criar seu novo **Álbum** e logo em seguida, criar um **Imagem**, assim:



Ao clicar no botão **"Salvar e continuar editando"** você pode conferir que a imagem foi salva com sucesso, e o arquivo para o campo **"original"** foi enviado e gravado no HD. Mas você também vai descobrir uma outra coisa: **onde está o thumbnail**?

Nós não fizemos nada a respeito. O que precisamos é descobrir uma maneira de criar a miniatura da imagem **original** e salvar no campo **"thumbnail"**.

Modificando o comportamento do Admin para criar a miniatura da imagem

Para isso, volte a abrir para edição o arquivo **"admin.py"** da pasta da aplicação **"galeria"**, e acrescente as seguintes linhas ao início do arquivo:

```
try:
    import Image
except ImportError:
    from PIL import Image


from django import forms
```

Como você já sabe, o **"try"** faz uma tentativa e se algo der errado, ele evita que o software pare ali mesmo e executa um **plano "B"**. O que estamos fazendo ali é **tentar** importar o pacote **"Image"**, que se trata da biblioteca de imagens **"PIL"**, mas pode haver uma diferença de versões, e por segurança, caso isso dê errado, uma segunda importação é feita como **plano "B"**.

O pacote **"forms"** do Django também é importado, para a criação de um novo formulário dinâmico.

Agora localize a seguinte linha no arquivo:

```
class AdminImagem(ModelAdmin):
```

Antes desta linha, você vai adicionar o seguinte trecho de código:

```
class FormImagem(forms.ModelForm):
    class Meta:
        model = Imagem


    def save(self, *args, **kwargs):
        """Metodo declarado para criar miniatura da imagem depois
de salvar"""
        imagem = super(FormImagem, self).save(*args, **kwargs)
```

```
        if 'original' in self.changed_data:
            extensao = imagem.original.name.split('.')[-1]
            imagem.thumbnail = 'galeria/thumbnail/%d.%s'%(
               imagem.id, extensao)

            miniatura = Image.open(imagem.original.path)
            miniatura.thumbnail((100,100), Image.ANTIALIAS)
            miniatura.save(imagem.thumbnail.path)

            imagem.save()

        return imagem
```

Vamos resumir o que fizemos:

O trecho abaixo declara uma classe de formulário dinâmico para a **entrada de dados** da classe de modelo **"Imagem"**. Observe que a palavra **"model"** está presente tanto na classe que herdamos quanto no atributo que indica a classe **"Imagem"**. Isso faz com que o formulário dinâmico tenha automaticamente todos os atributos da classe de modelo **"Imagem"** e ainda possui um método **"save()"** para **incluir** e **alterar** objetos dessa classe:

```
class FormImagem(forms.ModelForm):
    class Meta:
        model = Imagem
```

A seguir, vem a declaração do método **save()**. Como este método já existe na classe herdada, nós tomamos o cuidado de não interferir em seu funcionamento, e para isso nós declaramos os argumentos ***args** e ****kwargs**, que recebem os mesmos argumentos e repassam ao método da classe herdada, usando **"super()"**.

Dessa forma, o formulário dinâmico permanece funcionando exatamente em seu comportamento padrão, pois a função **"super()"** tem o papel de executar um método da forma como ele foi declarado na classe herdada (**"forms.ModelForm"**), e não na classe atual (**"FormImagem"**).

```
    def save(self, *args, **kwargs):
        """Metodo declarado para criar miniatura da imagem depois
de salvar"""
        imagem = super(FormImagem, self).save(*args, **kwargs)
```

Um pouco mais abaixo, fazemos uma interferência **após** a imagem ter sido

salva. A primeira linha define a condição de o campo **"original"** da imagem ter sido **alterado**. Isso é importante porque uma imagem pode ter seu título ou descrição alterados sem que o arquivo da imagem o seja.

Nós também extraímos a **extensão** do arquivo original e atribuímos o nome do novo arquivo de miniatura ao campo **"thumbnail"**:

```
        if 'original' in self.changed_data:
            extensao = imagem.original.name.split('.')[-1]
            imagem.thumbnail = 'galeria/thumbnail/%d.%s'%(
               imagem.id, extensao)
```

Em seguida, está a linha que **carrega** a imagem do arquivo original, usando a biblioteca **PIL**...

```
            miniatura = Image.open(imagem.original.path)
```

... cria sua miniatura com dimensões máximas de **100 x 100 pixels**...

```
            miniatura.thumbnail((100,100), Image.ANTIALIAS)
```

... e salva com o caminho completo do arquivo de miniatura que definimos:

```
            miniatura.save(imagem.thumbnail.path)
```

Se a imagem que você enviou é do tipo **JPEG**, o caminho completo, retornado por **"imagem.thumbnail.path"** será:

```
C:\Projetos\meu_blog\media\galeria\thumbnail\1.jpg
```

E para finalizar, a imagem é salva e retornada pelo método:

```
            imagem.save()


        return imagem
```

Agora lembre-se de que a classe **"AdminImagem"** não possui nenhum vínculo com o novo formulário dinâmico. Portanto, para dar efeito ao que fizemos, localize a seguinte linha:

```
    search_fields = ('titulo','descricao',)
```

E acrescente esta outra logo abaixo dela:

```
    form = FormImagem
```

Pronto! O arquivo **"admin.py"** completo ficou assim:

```
try:
    import Image
except ImportError:
    from PIL import Image
```

```
from django import forms
from django.contrib import admin
from django.contrib.admin.options import ModelAdmin

from models import Album, Imagem

class AdminAlbum(ModelAdmin):
    list_display = ('titulo',)
    search_fields = ('titulo',)

class FormImagem(forms.ModelForm):
    class Meta:
        model = Imagem

    def save(self, *args, **kwargs):
        """Metodo declarado para criar miniatura da imagem depois
de salvar"""
        imagem = super(FormImagem, self).save(*args, **kwargs)

        if 'original' in self.changed_data:
            extensao = imagem.original.name.split('.')[-1]
            imagem.thumbnail = 'galeria/thumbnail/%d.%s'%(
               imagem.id, extensao)

            miniatura = Image.open(imagem.original.path)
            miniatura.thumbnail((100,100), Image.ANTIALIAS)
            miniatura.save(imagem.thumbnail.path)

            imagem.save()

        return imagem

class AdminImagem(ModelAdmin):
```

```
    list_display = ('album','titulo',)
    list_filter = ('album',)
    search_fields = ('titulo','descricao',)
    form = FormImagem


admin.site.register(Album, AdminAlbum)
admin.site.register(Imagem, AdminImagem)
```

Salve o arquivo. Feche o arquivo. Faça o teste e veja como ficou!

A página de apresentação da galeria de imagens

Agora que temos todo o *back-end* da aplicação funcionando, vamos partir para o *front-end*.

Na pasta do projeto, abra o arquivo **"urls.py"** para edição e localize a seguinte linha:

```
    (r'^comments/', include('django.contrib.comments.urls')),
```

Esta linha faz a inclusão das URLs da aplicação **"comments"**, lembra-se? Pois bem, agora **abaixo** dela, escreva esta nova linha:

```
    (r'^galeria/', include('galeria.urls')),
```

Salve o arquivo. Feche o arquivo.

Vá até a pasta da aplicação **"galeria"** e crie um novo arquivo chamado **"urls.py"**, com o seguinte código dentro:

```
from django.conf.urls.defaults import *


urlpatterns = patterns('galeria.views',
    url(r'^$', 'albuns', name='albuns'),
)
```

Salve o arquivo. Feche o arquivo.

Agora, para responder a essa nova URL que criamos, crie outro novo arquivo, chamado **"views.py"**, com o seguinte código dentro:

```
from django.shortcuts import render_to_response
from django.template import RequestContext


from models import Album
```

```
def albuns(request):
    lista = Album.objects.all()
    return render_to_response(
        'galeria/albuns.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Salve o arquivo. Feche o arquivo.

Até aqui não há nada de novo: teremos uma URL **"/galeria/"** que vai exibir uma página do template **"galeria/albuns.html"** com uma lista de álbuns.

Agora, ainda na pasta da aplicação **"galeria"**, crie uma nova pasta chamada **"templates"** e dentro dela crie outra pasta chamada **"galeria"**. E agora, dentro dessa nova pasta (**"galeria/templates/galeria"**), crie o arquivo de template, chamado **"albuns.html"** com o seguinte código dentro:

```
{% extends "base.html" %}

{% block titulo %}Galeria de Imagens -
{{ block.super }} {% endblock %}

{% block h1 %}Galeria de imagens{% endblock %}

{% block conteudo %}

<ul>
    {% for album in lista %}
    <li>
    <a href="{{ album.get_absolute_url }}">{{ album }}</a>
    </li>
    {% endfor %}
</ul>

{% endblock conteudo %}
```

Salve o arquivo. Feche o arquivo.

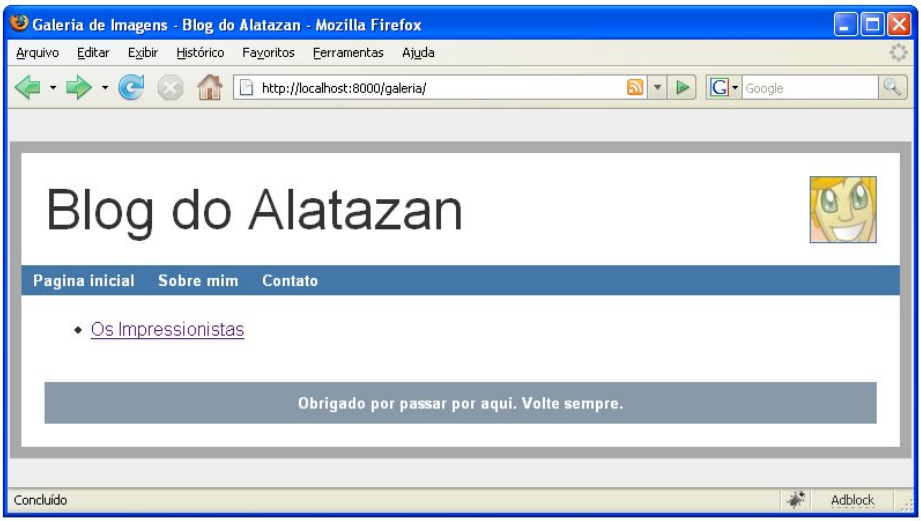
Aqui também não houve nada de novo: nós estendemos o template

"base.html" para trazer a página padrão do site, exibimos o **título da página** e fizemos um laço em todos os **Álbuns** da variável **"lista"**, exibindo o link para cada um em uma lista não-numerada (**<ul>**).

O conjunto que declaramos (a URL, a View e o Template) podem ser acessados agora no navegador pelo seguinte endereço:

```
http://localhost:8000/galeria/
```

Veja como ficou:



Agora precisamos criar uma página para o Álbum, certo? Pois vá até a pasta da aplicação **"galeria"**, abra novamente o arquivo **"urls.py"** para edição e acrescente esta nova URL:

```
    url(r'^(?P<slug>[\w_-]+)/$', 'album', name='album'),
```

Desta forma, o arquivo **"urls.py"** ficou assim:

```
from django.conf.urls.defaults import *


urlpatterns = patterns('galeria.views',
    url(r'^$', 'albuns', name='albuns'),
    url(r'^(?P<slug>[\w_-]+)/$', 'album', name='album'),
)
```

Salve o arquivo. Feche o arquivo.

Ainda na mesma pasta, abra o arquivo **"views.py"** para edição, e acrescente as

seguintes linhas de código ao final do arquivo:

```
def album(request, slug):
    album = get_object_or_404(Album, slug=slug)
    imagens = Imagem.objects.filter(album=album)

    return render_to_response(
        'galeria/album.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Esta nova **view** vai responder à URL do álbum, mas isso não é suficiente, pois precisamos importar a função **"get_object_or_404"**. Portanto localize a seguinte linha:

```
from django.shortcuts import render_to_response
```

E a modifique para ficar assim:

```
from django.shortcuts import render_to_response, get_object_or_404
```

Observe que o que fizemos na linha acima é o mesmo que isso:

```
from django.shortcuts import render_to_response
from django.shortcuts import get_object_or_404
```

Também precisamos importar a classe **"Imagem"**. Localize a seguinte linha:

```
from models import Album
```

E a modifique para ficar assim:

```
from models import Album, Imagem
```

Agora todo o arquivo **"views.py"** ficou assim:

```
from django.shortcuts import render_to_response, get_object_or_404
from django.template import RequestContext


from models import Album, Imagem


def albuns(request):
    lista = Album.objects.all()
    return render_to_response(
        'galeria/albuns.html',
```

```
        locals(),
        context_instance=RequestContext(request),
        )


def album(request, slug):
    album = get_object_or_404(Album, slug=slug)
    imagens = Imagem.objects.filter(album=album)

    return render_to_response(
        'galeria/album.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Salve o arquivo. Feche o arquivo.

Agora, partindo da pasta dessa aplicação, vá até a pasta **"templates/galeria"** e crie um novo arquivo chamado **"album.html"**, com o seguinte código dentro:

```
{% extends "base.html" %}


{% block titulo %}{{ album }} -
{{ block.super }} {% endblock %}

{% block h1 %}{{ album }}{% endblock %}

{% block conteudo %}
<div class="miniaturas">

    {% for imagem in imagens %}
    <div class="miniatura">
        <a href="{{ imagem.get_absolute_url }}">
        <img src="{{ imagem.thumbnail.url }}"/>
        {{ imagem }}
        </a>
    </div>
```

```
    {% endfor %}


</div>
{% endblock conteudo %}
```

Veja que continuamos trabalhando em cima do que já conhecemos. Dê uma atenção especial às referências **"{{ album }}"** e **"{{ imagem.thumbnail.url }}"**. Observe que a URL do campo **"thumbnail"** é acessível através do seu atributo **"url"**.

Salve o arquivo. Feche o arquivo.

Agora, antes de testar, volte à pasta da aplicação **"galeria"**, abra o arquivo **"models.py"** para edição e localize a seguinte linha:

```
from django.db import models
```

Acrescente a seguinte linha abaixo dela:

```
from django.core.urlresolvers import reverse
```

Agora, ao final da classe **"Album"**, acrescente o seguinte trecho de código:

```
def get_absolute_url(self):
    return reverse('album', kwargs={'slug': self.slug})
```

E faça o mesmo para a classe **"Imagem"**. Observe a diferença entre as duas - a palavra **"imagem"** como primeiro argumento da função **"reverse()"**, é o nome que demos a essas URLs, usando seu argumento **"name"**, lá atrás, no arquivo **"urls.py"**:

```
def get_absolute_url(self):
    return reverse('imagem', kwargs={'slug': self.slug})
```

Agora o arquivo **"models.py"** ficou assim:

```
from datetime import datetime
from django.db import models
from django.core.urlresolvers import reverse


class Album(models.Model):
    """Um album eh um pacote de imagens, ele tem um titulo e um
slug para sua identificacao."""
    class Meta:
        ordering = ('titulo',)


    titulo = models.CharField(max_length=100)
```

```
    slug = models.SlugField(max_length=100, blank=True,
unique=True)


    def __unicode__(self):
        return self.titulo


    def get_absolute_url(self):
        return reverse('album', kwargs={'slug': self.slug})


class Imagem(models.Model):
    """Cada instancia desta classe contem uma imagem da galeria,
com seu respectivo thumbnail (miniatura) e imagem em tamanho
natural. Varias imagens podem conter dentro de um Album"""


    class Meta:
        ordering = ('album','titulo',)


    album = models.ForeignKey('Album')
    titulo = models.CharField(max_length=100)
    slug = models.SlugField(
    max_length=100,
    blank=True,
    unique=True
    )
    descricao = models.TextField(blank=True)
    original = models.ImageField(
        null=True,
        blank=True,
        upload_to='galeria/original',
        )
    thumbnail = models.ImageField(
        null=True,
        blank=True,
        upload_to='galeria/thumbnail',
        )
```

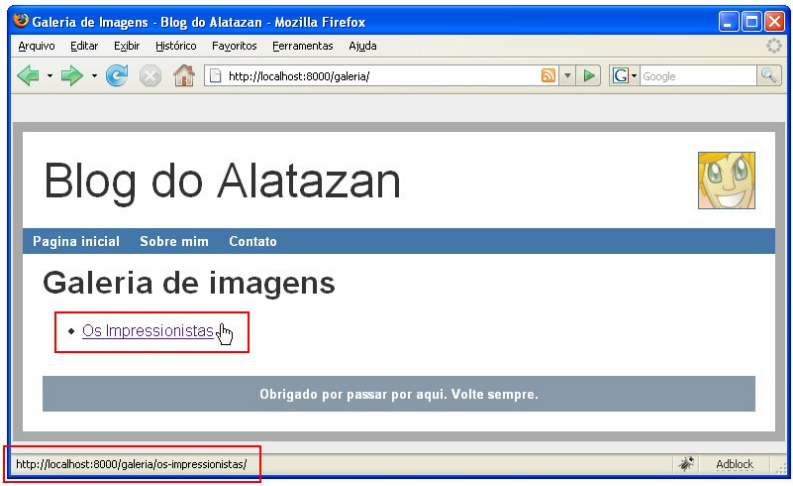
```
    publicacao = models.DateTimeField(
    default=datetime.now,
    blank=True
    )

    def __unicode__(self):
        return self.titulo

    def get_absolute_url(self):
        return reverse('imagem', kwargs={'slug': self.slug})

# SIGNALS
from django.db.models import signals
from utils.signals_comuns import slug_pre_save

signals.pre_save.connect(slug_pre_save, sender=Album)
signals.pre_save.connect(slug_pre_save, sender=Imagem)
```

Salve o arquivo. Feche o arquivo.

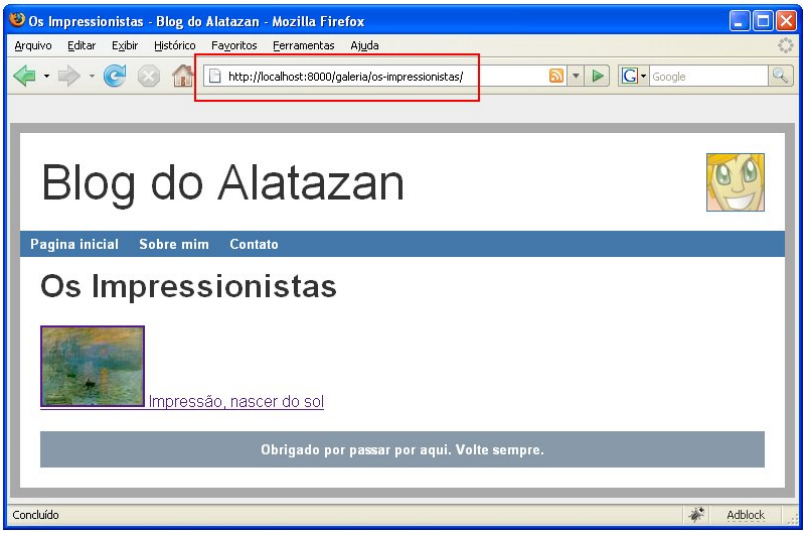
No navegador, carregue a URL de nossa galeria de imagens:

```
http://localhost:8000/galeria/
```

Veja como aparece:



Agora clique sobre o **link** do álbum e veja como a página é carregada:



Estamos agora com as miniaturas do álbum na tela! Agora, resta criar a página para visualizar a foto em seu tamanho original!

Criando uma página só para a imagem em seu tamanho original

Na pasta da aplicação **"galeria"**, abra o arquivo **"urls.py"** para edição, e acrescente esta nova URL:

```
    url(r'^imagem/(?P<slug>[\w_-]+)/$', 'imagem', name='imagem'),
```

Agora, o arquivo **"urls.py"** deve ficar assim:

```
from django.conf.urls.defaults import *


urlpatterns = patterns('galeria.views',
    url(r'^$', 'albuns', name='albuns'),
    url(r'^(?P<slug>[\w_-]+)/$', 'album', name='album'),
    url(r'^imagem/(?P<slug>[\w_-]+)/$', 'imagem', name='imagem'),
)
```

Salve o arquivo. Feche o arquivo.

Agora abra o arquivo **"views.py"** da mesma pasta para edição, e acrescente o trecho de código abaixo ao final do arquivo:

```
def imagem(request, slug):
```

```
    imagem = get_object_or_404(Imagem, slug=slug)


    return render_to_response(
        'galeria/imagem.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Como pode ver, esta nova **view** é muito semelhante à do **Álbum**.

Salve o arquivo. Feche o arquivo.

Agora, ainda na pasta da aplicação, vá até a pasta **"templates/galeria"** e crie um novo arquivo chamado **"imagem.html"**, com o seguinte código dentro:

```
{% extends "base.html" %}


{% block titulo %}{{ imagem }} - {{ block.super }}{% endblock %}


{% block h1 %}{{ imagem }}{% endblock %}


{% block conteudo %}
<div class="imagem">
    <img src="{{ imagem.original.url }}"/>


    {{ imagem.descricao|linebreaks }}


</div>
{% endblock conteudo %}
```

Como pode ver, seguimos a mesma *cartilha* também para este template.

Salve o arquivo. Feche o arquivo.

Agora volte ao navegador, atualize a página com **F5** e clique sobre a miniatura da imagem. Veja que a nova página é carregada:

Uau! Temos uma batalha quase vencida! Agora é só colocar um item no menu!

Ajustando o template base para a nova aplicação

Na pasta **"templates"** do projeto, abra o arquivo **"base.html"** para edição e localize a seguinte linha:

```
<li><a href="/">Pagina inicial</a></li>
```

Logo abaixo da linha encontrada, acrescente esta outra:

```
<li>
<a href="{% url galeria.views.albuns %}">Galeria de imagens
</a>
</li>
```

Salve o arquivo. Feche o arquivo. Volte ao navegador e atualize, com a tecla **F5**. Veja agora como ficou:

Pronto! E assim temos mais uma página virada, ou várias páginas, né?

Agora vamos organizar as coisas!

Alatazan ficou fascinado! Não parecia possível criar uma aplicação de imagens em tão pouco tempo!

- É fascinante! Resumindo o que fizemos não foi muita coisa:
  - Criamos uma nova aplicação;
  - Criamos duas classes, uma relacionada à outra, usando um campo de **ForeignKey**;
  - Definimos a **pasta para upload** dos arquivos para os campos de imagem;
  - Fizemos um *refactoring* no signal que gera slugs;
  - Instalamos a aplicação no projeto, usando a setting **"INSTALLED_APPS"**;
  - Instalamos a **PIL** para o Django trabalhar com imagens;
  - Geramos as **novas tabelas** no banco de dados;
  - Criamos um Admin melhor elaborado, com alguns campos na **lista**, **filtros** e **busca**;
  - Ajustamos o Admin para criar as miniaturas para o campo **"thumbnail"**... isso foi feito com um **ModelForm**;

- E criamos os **templates** para a aplicação.

- É... pensando melhor, foram muitas coisas sim...

- E olha, Alatazan, foram muitas coisas que aprendemos, mas é importante você **observar com atenção** o código dos arquivos que criamos, pois há ali muitas informações importantes no aprendizado. O Django é muito claro, muito coeso, as palavras usadas para identificar qualquer coisa possuem um significado forte... - Nena fez sua (quase) advertência.

- T...

Bláaaaaa!!!

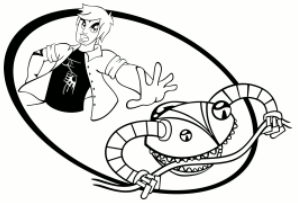
O barulho foi forte, e veio da biblioteca do pai de Cartola... Alatazan presumiu que seria alguma coisa relacionada ao **Ballzer**... ele estava tão comportado nos últimos dias que esse tipo de "comportado" não daria em coisa boa...

E ele presumiu com razão... Ballzer ainda estava um pouco atordoado com a situação, e o carpete verde-escuro tinha todos os livros espalhados... era fácil notar um livro sobre **"ervas medicinais"** enfiado ao meio de outro sobre **"raizes culturais"**.

Alatazan sabia que por mais íntimas que fossem as palavras **"ervas"** e **"raízes"**, não havia nada ali de comum entre os dois **temas**...

... e à medida que se desculpava e tentava dividir sua atenção entre juntar os livros e brigar com Ballzer, ele descobriu **mais uma coisa** pra colocar em seu site...

Capítulo 21: Organizando as coisas com Tags



O céu de Katara é pintado de cores **variadas**.

Os antepassados acreditavam que no alto da montanha mais alta, havia um alto senhor, de olhos altivos, cabelos alongados, barbas brancas e compridas, que calçava sandálias brancas e usava um vestido também comprido e branco.

Um dia aquele alto homem, de olhos altivos, cabelos alongados, barbas brancas e compridas, que calçava sempre as mesmas sandálias das mesmas cores e tudo muito branco, alto e comprido, se encheu de tudo isso e comprou **óculos coloridos**.

Ninguém nunca soube explicar onde ele encontrou aqueles óculos coloridos para comprar, mas como a história fala sobre a altivez do homem e na virada que isso causou em sua vida e sua coluna, isso não nos importa agora.

O que realmente importa é que naquele dia o homem se rebelou e criou uma imensa tecnologia, dotada de jatos ultra-potentes de tinta, que fizeram o céu se colorir e o povo das planícies protestar contra a quantidade de tinta que caía lá embaixo.

Dizem que é por isso que todos os katarenses hoje sabem que entre o branco e o preto há **muitas cores**, e algumas delas podem estar misturadas em sua cabeça. Não se esqueça de lavar com shapoo Ultrapoo Limpadon, aquele que espalha, mistura e lustra, e depois que seca, também brilha.

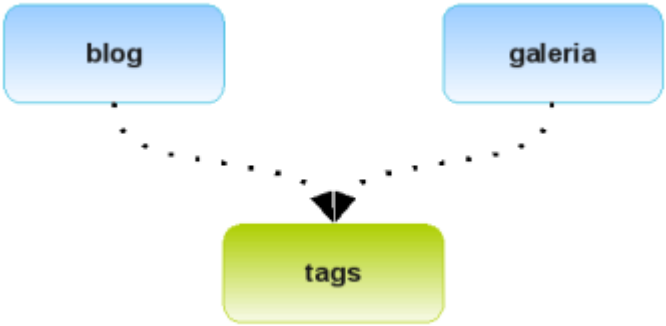
Mas o sabão que o Alatazan deu no Ballzer não foi suficiente. Eles precisavam organizar agora tudo novamente em seus **devidos assuntos**... e na vida real, não é possível ter um mesmo livro em **duas seções ao mesmo tempo**... por sorte, no site seria mais fácil...

Separando as coisas com etiquetas variadas

Tags são etiquetas, uma forma de organizar as informações na Web que foi massificada pela Web2.0, especialmente pelo site **del.icio.us**.

A idéia de usar *tags* geralmente é assim: um campo de texto simples onde o usuário informa as **palavras-chave** - as tags - daquele objeto, quantas palavras ele quiser informar. Ao salvar, o objeto é vinculado a todas aquelas palavras, e então através de outra página é possível ver todas essas Tags, clicar sobre elas e saber quais objetos estão ligados a elas.

Nós podemos ter *tags* em **artigos do blog** e em **imagens da galeria**. Com isso, as aplicações **"blog"** e **"galeria"** passam a ser dependentes da nova aplicação que vamos criar: **"tags"**. Mas por outro lado, a aplicação **"tags"** não será dependente de ninguém.



Então, *let's go*!

Criando a nova aplicação de tags

Na pasta do projeto, crie uma pasta para a nova aplicação, chamada **"tags"** e dentro dela crie um arquivo vazio chamado **"__init__.py"**.

Agora crie um novo arquivo chamado **"models.py"**, com o seguinte código dentro:

```
from django.db import models
from django.contrib.contenttypes.models import ContentType


class Tag(models.Model):
    nome = models.CharField(max_length=30, unique=True)


    def __unicode__(self):
        return self.nome
```

```
class TagItem(models.Model):
    class Meta:
        unique_together = ('tag', 'content_type', 'object_id')


    tag = models.ForeignKey('Tag')
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField(db_index=True)
```

Você pode ver que temos duas classes de modelo: **"Tag"** e **"TagItem"**, e que a classe **"TagItem"** está relacionada à classe **"Tag"** de forma que uma **tag** pode conter muitos **itens**.

Mas você agora vai descobrir essa **importação** diferente do que aprendemos até aqui:

```
from django.contrib.contenttypes.models import ContentType
```

A aplicação de *contrib* **"contenttypes"** tem a função de trabalhar com **tipos dinâmicos**, ou melhor dizendo: ela permite que tipos **que não se conhecem** sejam relacionados uns aos outros.

Na prática, usando a classe de modelo que importamos na linha acima (**"ContentType"**) é possível ter um **artigo** do **blog** vinculado a uma **tag** sem que eles se conheçam, mantendo uma dependência mínima, apenas em um lugar do código-fonte, sem precisar de alterar o banco de dados para isso.

A seguinte linha possui um argumento importante: **"unique=True"**. Ele vai garantir que não teremos tags **duplicadas**:

```
    nome = models.CharField(max_length=30, unique=True)
```

Já a linha abaixo tem o papel de garantir que a classe de modelo (neste caso, **"TagItem"**) terá somente um único objeto para cada combinação dos três campos informados: **"tag"**, **"content_type"** e **"object_id"**.

Por exemplo: um objeto de **"TagItem"** com a tag **"livros"**, o content_type **"blog.artigo"** e o object_id **15** só pode existir uma única vez.

```
    unique_together = ('tag', 'content_type', 'object_id')
```

Agora observe este trecho de código abaixo. Traduzindo, a combinação desses campos com o campo **"tag"** permite que seja possível ligar uma **tag** a qualquer objeto do projeto.

```
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField(db_index=True)
```

O campo **"content_type"** representa um **ContentType** - o tipo dinâmico. Ele pode ser **"blog.artigo"** (classe **"Artigo"** da aplicação **"blog"**) ou então

"galeria.imagem" (classe **"Imagem"** da aplicação **"galeria"**). Na verdade pode ser qualquer outro tipo que exista no projeto.

E já o campo **"object_id"** representa o código **id** do objeto relacionado.

Exemplo:

- tag: **"livro"**
- content_type: **"blog.artigo"**
- object_id: **15**

O exemplo acima indica que o **artigo do blog** de id **15** possui a tag **"livro"**.

O argumento **"db_index=True"** cria um índice no banco de dados para o campo **"object_id"**, o que deve tornar as nossas buscas mais rápidas quando o relacionamento for feito diretamente a ele.

Salve o arquivo. Feche o arquivo.

Agora precisamos instalar a nova aplicação no projeto, certo? Então na pasta do projeto, abra o arquivo **"settings.py"** para edição e acrescente a seguinte linha à setting **"INSTALLED_APPS"**:

```
    'tags',
```

Agora a setting que modificamos ficou assim:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.syndication',
    'django.contrib.flatpages',
    'django.contrib.comments',


    'blog',
    'galeria',
    'tags',
)
```

Salve o arquivo. Feche o arquivo.

Clique duas vezes no arquivo **gerar_banco_de_dados.bat** para gerar as novas

tabelas no banco de dados. O resultado será este:

```
Creating table tags_tag
Creating table tags_tagitem
Installing index for tags.Tag model
Installing index for tags.TagItem model
```

Feche a janela do **MS-DOS**.

Agora vamos modificar a aplicação **"blog"** para que a classe **"Artigo"** permita a entrada de **tags**. Para isso, vá até a pasta da aplicação **"blog"** e abra o arquivo **"admin.py"** para edição. Logo abaixo da primeira linha, acrescente o trecho de código abaixo:

```
from django.contrib.admin.options import ModelAdmin
from django import forms
```

Você já conhece essas duas importações. Lembra-se do último capítulo? A classe **"ModelAdmin"** permite a criação de um Admin mais elaborado para uma classe. Já o pacote **"forms"** permite a criação de **formulários dinâmicos**, e o ModelAdmin faz uso deles!

Agora encontre esta outra linha:

```
from models import Artigo
```

Acrescente abaixo dela o seguinte trecho de código:

```
class FormArtigo(forms.ModelForm):
    class Meta:
        model = Artigo


    tags = forms.CharField(max_length=30, required=False)
```

Esse formulário dinâmico vai representar a entrada de dados na classe ModelAdmin de **"Artigo"**, e além de todas as atribuições que ele já possui, ele vai ter um campo **calculado** a mais, chamado **"tags"**.

Campos calculados são campos que não existem de fato no banco de dados, mas que são úteis para alguma coisa ser tratada em **memória**.

Agora abaixo do trecho de código que escrevemos, acrescente mais este:

```
class AdminArtigo(ModelAdmin):
    form = FormArtigo
```

Aqui nós demos vida ao formulário dinâmico **"FormArtigo"**, vinculando-o ao ModelAdmin que será registrado para a classe **"Artigo"**.

E por fim, **modifique** a última linha:

```
admin.site.register(Artigo)
```

Para ficar assim:

```
admin.site.register(Artigo, AdminArtigo)
```

Com as mudanças que fizemos, o arquivo **"admin.py"** ficou assim:

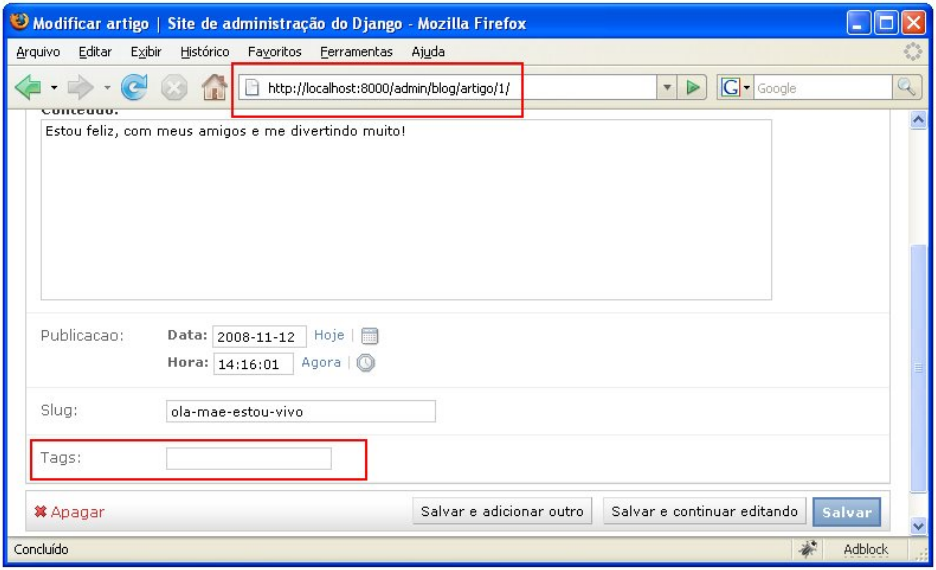
```
from django.contrib import admin
from django.contrib.admin.options import ModelAdmin
from django import forms

from models import Artigo

class FormArtigo(forms.ModelForm):
    class Meta:
        model = Artigo

    tags = forms.CharField(max_length=30, required=False)

class AdminArtigo(ModelAdmin):
    form = FormArtigo

admin.site.register(Artigo, AdminArtigo)
```

Salve o arquivo. Feche o arquivo.

Agora vamos executar o projeto para ver o efeito do que fizemos. Na pasta do projeto, execute o arquivo **"executar.bat"**, abra seu navegador e carregue a seguinte URL:

```
http://localhost:8000/admin/blog/artigo/1/
```

Essa URL vai abrir o **artigo** de código **1** no Admin. Veja:



No entanto, ao informar algum valor ali e **salvar**, isso não terá efeito nenhum, porque trata-se de um campo de memória, sem uma representação **persistente** no banco de dados.

Para resolver isso, volte a abrir o arquivo **"admin.py"** da aplicação **"blog"** para edição e localize a seguinte linha:

```
    tags = forms.CharField(max_length=30, required=False)
```

Abaixo dela, acrescente o seguinte trecho de código:

```
    def save(self, *args, **kwargs):
        artigo = super(FormArtigo, self).save(*args, **kwargs)

        aplicar_tags(artigo, self.cleaned_data['tags'])

        return artigo
```

Você também já conhece o método **"save()"** e sabe que a função **"super()"** retorna para a variável **"artigo"** o objeto salvo pelo mesmo método na classe herdada (**"forms.ModelForm"**). No entanto, a seguinte linha é novidade para você:

```
        aplicar_tags(artigo, self.cleaned_data['tags'])
```

Esta linha chama a função **"aplicar_tags()"**, passando como seus argumentos: o artigo salvo e o conteúdo do campo **"tags"**, informado pelo usuário.

Todo formulário dinâmico possui o atributo **"cleaned_data"**, que é um dicionário que traz todos os campos do formulário, contendo o valor informado pelo usuário.

Já quanto à função **"aplicar_tags()"**, nós vamos importá-la agora mesmo da aplicação **"tags"**. Para fazer isso, localize a seguinte linha:

```
from models import Artigo
```

E acrescente esta abaixo dela:

```
from tags import aplicar_tags
```

Agora após essas modificações, o arquivo **"admin.py"** ficou assim:

```
from django.contrib import admin
from django.contrib.admin.options import ModelAdmin
from django import forms

from models import Artigo
from tags import aplicar_tags

class FormArtigo(forms.ModelForm):
    class Meta:
        model = Artigo

    tags = forms.CharField(max_length=30, required=False)

    def save(self, *args, **kwargs):
        artigo = super(FormArtigo, self).save(*args, **kwargs)

        aplicar_tags(artigo, self.cleaned_data['tags'])

        return artigo

class AdminArtigo(ModelAdmin):
    form = FormArtigo

admin.site.register(Artigo, AdminArtigo)
```

Salve o arquivo. Feche o arquivo.

Agora precisamos criar a função **"aplicar_tags()"**, né?

Vá até a pasta da aplicação **"tags"** e abra o arquivo **"__init__.py"** para edição. Acrescente as seguintes linhas a ele:

```
from models import Tag, TagItem
from django.contrib.contenttypes.models import ContentType


def aplicar_tags(obj, tags):
    tipo_dinamico = ContentType.objects.get_for_model(obj)


    TagItem.objects.filter(
        content_type=tipo_dinamico,
        object_id=obj.id,
        ).delete()


    tags = tags.split(' ')
    for tag_nome in tags:
        tag, nova = Tag.objects.get_or_create(nome=tag_nome)


        TagItem.objects.get_or_create(
            tag=tag,
            content_type=tipo_dinamico,
            object_id=obj.id,
            )
```

Observe que a classe **"Artigo"** não é informada, nem sequer a aplicação **"blog"** é citada.

Pois vamos agora analisar o que fizemos por partes:

As duas importações abaixo, você já conhece: são as classes das quais precisamos para trabalhar a função que vai **aplicar as tags** informadas no formulário do **artigo** ou de qualquer outro objeto.

```
from models import Tag, TagItem
from django.contrib.contenttypes.models import ContentType
```

Aqui nós declaramos a função. Ela possui dois argumentos: o **objeto** e as **tags** que serão aplicadas a ele:

```
def aplicar_tags(obj, tags):
```

A próxima coisa que fazemos é carregar o **tipo dinâmico** para o objeto que estamos tratando. É aqui que o Django descobre se o objeto é **Artigo**, **Imagem** ou seja lá o que for. O valor atribuído à variável **"tipo_dinamico"** será o **ContentType** que representa a classe de modelo do objeto informado pela variável **"obj"**:

```
tipo_dinamico = ContentType.objects.get_for_model(obj)
```

Após carregar o tipo dinâmico para o objeto que estamos trabalhando, é a hora de **excluir** todas as tags que o objeto possui. As linhas a seguir carregam todos os objetos de **"TagItem"** do tipo dinâmico do objeto e de seu código (**"obj.id"**). Depois disso, exclui a todos com o método **".delete()"**:

```
TagItem.objects.filter(
      content_type=tipo_dinamico,
      object_id=obj.id,
      ).delete()
```

Fazemos isso porque um pouco mais pra baixo vamos criá-las novamente, garantindo que ficarão para o objeto exatamente as tags da variável **"tags"**.

Mas **atenção**: o trecho de código acima não exclui as tags em si, mas sim o vínculo delas com o objeto.

Agora, na linha baixo, nós **separamos** por espaços (**' '**) a string das tags informadas pelo usuário. Ou seja, se o usuário houver informado **"arte brasil"**, o resultado será **['arte', 'brasil']**:

```
tags = tags.split(' ')
```

Logo a seguir, fazemos um laço nas tags, ou seja, no caso citado acima (**['arte', 'brasil']**), esse laço irá executar seu bloco duas vezes: uma para a palavra **"arte"**, outra para a palavra **"brasil"**. A string da tag na execução do bloco está contida na variável **"tag_nome"**:

```
for tag_nome in tags:
```

Agora, dentro do bloco, a linha abaixo tenta carregar um objeto do banco de dados para a tag atual dentro do laço. Se não existir no banco de dados, ela então é criada, e a variável **"nova"** é atribuída com valor **True**. Isso é feito especificamente pelo método **"get_or_create()"**. Já a variável **"tag"** recebe a tag carregada do banco de dados, seja ela nova ou não:

```
      tag, nova = Tag.objects.get_or_create(nome=tag_nome)
```

E por fim, temos o bloco que cria o vínculo de cada tag com o objeto, também seguindo a mesma lógica de pensamento, criando somente se existir, para evitar um eventual conflito:

```
      TagItem.objects.get_or_create(
```

```
            tag=tag,
            content_type=tipo_dinamico,
            object_id=obj.id,
            )
```

Puxa, quanto falatório!

Salve o arquivo. Feche o arquivo. Volte ao navegador, na URL do artigo no Admin:

```
http://localhost:8000/admin/blog/artigo/1/
```

Informe as tags **"arte brasil"** e clique no botão de **"Salvar e continuar editando"**. No entanto, vai perceber que não adiantou nada, ele parece não salvar.

Acontece que o nosso formulário dinâmico **"FormArtigo"** possui o tratamento para **salvar** o campo **"tags"** usando a função **"aplicar_tags"**, mas não possui o tratamento para **carregá-las** depois de salvo. Percebeu a diferença? Nós estamos gravando no banco de dados, mas isso não está sendo mostrado na tela, e como o campo **"tags"** é um campo calculado, ele vai mostrar somente aquilo que nós, manualmente, definirmos para ele mostrar.

Portanto, agora volte ao arquivo **"admin.py"** da aplicação **"blog"** para edição, e localize esta linha:

```
    tags = forms.CharField(max_length=30, required=False)
```

Abaixo dela, acrescente o seguinte trecho de código:

```
    def __init__(self, *args, **kwargs):
        super(FormArtigo, self).__init__(*args, **kwargs)


        if self.instance.id:
            self.initial['tags'] = tags_para_objeto(self.instance)
```

Este método, que é o **inicializador** do formulário dinâmico, é executado toda vez que o formulário é instanciado, seja para exibir dados na tela, seja para salvar os dados informados pelo usuário. E é exatamente aqui, depois de executar o inicializador da classe herdada (com a função **"super()"**) que ele carrega o valor para o campo calculado **"tags"**:

```
        if self.instance.id:
            self.initial['tags'] = tags_para_objeto(
            self.instance
            )
```

A linha acima tem o seguinte significado: se o formulário possui uma **instância**

de objeto, o valor **inicial** para o campo **"tags"** deve ser o que é retornado pela função **"tags_para_objeto()"**.

E falando na função, ainda precisamos trazê-la de algum lugar, certo? Localize a seguinte linha:

```
from tags import aplicar_tags
```

E a modifique, para ficar assim:

```
from tags import aplicar_tags, tags_para_objeto
```

Pronto, agora desta forma o Admin da classe **"Artigo"** deve funcionar corretamente.

Salve o arquivo. Feche o arquivo.

E como declaramos a função **"tags_para_objeto()"**, precisamos agora que ela exista. Então, na pasta da aplicação **"tags"**, abra o arquivo **"__init__.py"** para edição e acrescente o trecho de código abaixo ao seu final:

```
def tags_para_objeto(obj):
    tipo_dinamico = ContentType.objects.get_for_model(obj)


    tags = TagItem.objects.filter(
        content_type=tipo_dinamico,
        object_id=obj.id,
        )


    return ' '.join([item.tag.nome for item in tags])
```

Aí está a nossa função!

Veja que a primeira linha carrega o tipo dinâmico para o objeto:

```
  tipo_dinamico = ContentType.objects.get_for_model(obj)
```

E aqui nós carregamos a lista de **"TagItem"** para o objeto (tipo dinâmico + o **id** do objeto):

```
    tags = TagItem.objects.filter(
        content_type=tipo_dinamico,
        object_id=obj.id,
   )
```

E por fim, bom... na última linha nós temos uma *list comprehension* das tags para transformá-las em uma string separada por espaços... mas vamos refletir um pouco para compreender como isso funciona...

Isto é uma *list comprehension*:

```
[item for item in tags]
```

A variável **"tags"** possui uma lista de objetos do tipo **"TagItem"**, são os vínculos que temos entre o objeto em questão (o artigo "Olá mãe! Estou vivo!") e as tags "arte" e "brasil".

A *list comprehension* acima, criada sobre a variável **"tags"**, retorna a variável exatamente como ela é, pois retornamos um a um de seus itens, sem nenhuma interferência, veja:

```
[<TagItem: TagItem object>, <TagItem: TagItem object>]
```

Mas nós podemos retornar o nome das tags no lugar dos objetos, assim:

```
[item.tag.nome for item in tags]
```

E o resultado será este:

```
[u'arte', u'brasil']
```

É para isso que *list comprehensions* existem: retornar o conteúdo de uma lista com algumas interferências que nós desejamos.

Agora, quando se usa o método **".join()"** em uma string, como abaixo:

```
' --- '.join([u'arte', u'brasil'])
```

Veja qual é o resultado disso:

```
u'arte --- brasil'
```

Note que o **' --- '** é usado como separador entre os ítens da lista, sejam lá quantos eles forem, veja este outro exemplo:

```
', '.join([u'arte', u'brasil', 'outra string', 'mais uma'])
```

E aqui o resultado:

```
u'arte, brasil, outra string, mais uma'
```

Veja que **', '** está entre cada uma das strings da lista.

Portanto, a última linha que escrevemos em nosso código:

```
  return ' '.join([item.tag.nome for item in tags])
```

Retornaria algo como isso:

```
u'arte brasil'
```

Ignore o **"u'"**, ele é apenas um indicador de que a string está em **formato Unicode**.

E o arquivo **"admin.py"** da aplicação **"blog"** acabou ficando assim:

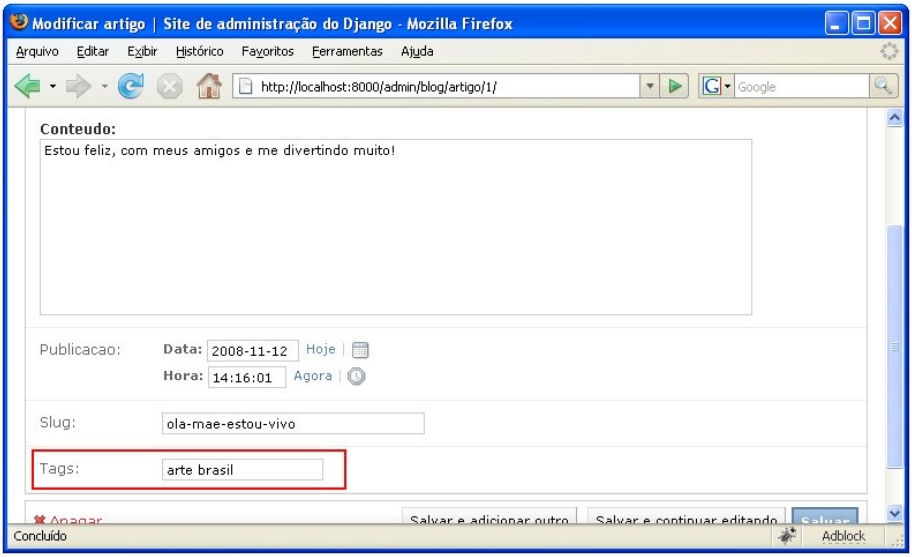
```
from django.contrib import admin
```

```
from django.contrib.admin.options import ModelAdmin
from django import forms

from models import Artigo
from tags import aplicar_tags, tags_para_objeto

class FormArtigo(forms.ModelForm):
    class Meta:
        model = Artigo

    tags = forms.CharField(max_length=30, required=False)

    def __init__(self, *args, **kwargs):
        super(FormArtigo, self).__init__(*args, **kwargs)

        if self.instance.id:
            self.initial['tags'] = tags_para_objeto(
            self.instance
            )

    def save(self, *args, **kwargs):
        artigo = super(FormArtigo, self).save(*args, **kwargs)

        aplicar_tags(artigo, self.cleaned_data['tags'])

        return artigo

class AdminArtigo(ModelAdmin):
    form = FormArtigo

admin.site.register(Artigo, AdminArtigo)
```

Salve o arquivo. Feche o arquivo. Volte ao navegador e carregue a página de Admin do nosso artigo novamente (mesmo que ele já esteja carregado na tela):

```
http://localhost:8000/admin/blog/artigo/1/
```

E agora, mesmo sem você ter salvo nada, a página é exibida com os valores corretos:



Fazendo a mesma coisa com as imagens da galeria

Agora, abra o arquivo **"admin.py"** da aplicação **"galeria"** e o modifique, para ficar assim:

```
try:
    import Image
except ImportError:
    from PIL import Image


from django import forms
from django.contrib import admin
from django.contrib.admin.options import ModelAdmin


from models import Album, Imagem
from tags import aplicar_tags, tags_para_objeto


class AdminAlbum(ModelAdmin):
```

```
    list_display = ('titulo',)
    search_fields = ('titulo',)


class FormImagem(forms.ModelForm):
    class Meta:
        model = Imagem


    tags = forms.CharField(max_length=30, required=False)


    def __init__(self, *args, **kwargs):
        super(FormImagem, self).__init__(*args, **kwargs)


        if self.instance.id:
            self.initial['tags'] = tags_para_objeto(
            self.instance
            )


    def save(self, *args, **kwargs):
        """Metodo declarado para criar miniatura da imagem depois
de salvar"""
        imagem = super(FormImagem, self).save(*args, **kwargs)


        if 'original' in self.changed_data:
            extensao = imagem.original.name.split('.')[-1]
            imagem.thumbnail = 'galeria/thumbnail/%d.%s'%(
            imagem.id, extensao
            )


            miniatura = Image.open(imagem.original.path)
            miniatura.thumbnail((100,100), Image.ANTIALIAS)
            miniatura.save(imagem.thumbnail.path)


            imagem.save()
```

```
        aplicar_tags(imagem, self.cleaned_data['tags'])


        return imagem


class AdminImagem(ModelAdmin):
    list_display = ('album','titulo',)
    list_filter = ('album',)
    search_fields = ('titulo','descricao',)
    form = FormImagem


admin.site.register(Album, AdminAlbum)
admin.site.register(Imagem, AdminImagem)
```

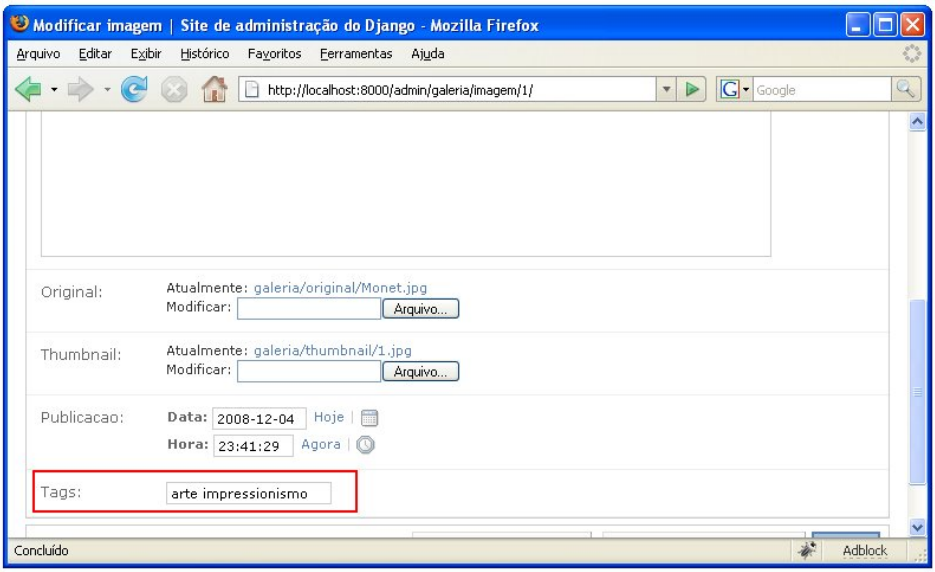
Observe que as novidades são:

1. Importamos as funções **"aplicar_tags()"** e **"tags_para_objeto()"**;
2. Acrescentamos o campo calculado **"tags"** ao formulário dinâmico da imagem;
3. Acrescentamos o método inicializador para carregar as tags;
4. Acrescentamos uma linha ao método **"save()"** para aplicar as tags à imagem;

Salve o arquivo. Feche o arquivo. Agora vá ao navegador e carregue a URL de uma imagem no Admin, esta por exemplo:

```
http://localhost:8000/admin/galeria/imagem/1/
```

Informe algumas palavras no campo **"tags"** e clique sobre o botão **"Salvar e continuar editando"**. Tudo funcionando uma beleza:



Agora criando uma página para as tags

Agora precisamos de uma página para listar as tags que existem em nosso site. Vamos começar pela URL: abra o arquivo **"urls.py"** da pasta do projeto e acrescente esta nova URL:

```
    (r'^tags/', include('tags.urls')),
```

Resumindo: criamos uma URL **"^tags/"** para todas as URLs da aplicação **"tags"**:

Agora o arquivo ficou assim:

```
from django.conf.urls.defaults import *
from django.conf import settings

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

from blog.models import Artigo
from blog.feeds import UltimosArtigos
```

```
urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index',
        {'queryset': Artigo.objects.all(),
         'date_field': 'publicacao'}),
    (r'^admin/(.*)', admin.site.root),
    (r'^rss/(?P<url>.*)/$',
     'django.contrib.syndication.views.feed',
        {'feed_dict': {'ultimos': UltimosArtigos}}),
    (r'^artigo/(?P<slug>[\w_-]+)/$', 'blog.views.artigo'),
    (r'^contato/$', 'views.contato'),
    (r'^comments/', include('django.contrib.comments.urls')),
    (r'^galeria/', include('galeria.urls')),
    (r'^tags/', include('tags.urls')),
)

if settings.LOCAL:
    urlpatterns += patterns('',
        (r'^media/(.*)$', 'django.views.static.serve',
         {'document_root': settings.MEDIA_ROOT}),
    )
```

Salve o arquivo. Feche o arquivo.

Agora vá à pasta da aplicação **"tags"** e crie um novo arquivo **"urls.py"** com o seguinte código dentro:

```
from django.conf.urls.defaults import *


urlpatterns = patterns(
    'tags.views',
    url(r'^$', 'tags', name='tags'),
)
```

Salve o arquivo. Feche o arquivo.

Agora precisamos criar a view que indicamos à URL acima. Para isso crie outro novo arquivo na pasta da aplicação **"tags"**, chamado **"views.py"**, com o seguinte código dentro:

```
from django.shortcuts import render_to_response
```

```
from django.template import RequestContext


from models import Tag


def tags(request):
    lista = Tag.objects.all()
    return render_to_response(
        'tags/tags.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Ali temos a view da URL **"/tags/"**, que retorna uma **lista de tags**, usando o template **"tags/tags.html"**.

Salve o arquivo. Feche o arquivo. Agora, vamos criar o template!

Ainda na pasta da aplicação **"tags"**, crie a pasta **"templates"** e dentro dela outra pasta chamada **"tags"**. Agora dentro da nova pasta crie o arquivo **"tags.html"** com o seguinte código dentro:

```
{% extends "base.html" %}


{% block titulo %}Tags - {{ block.super }}{% endblock %}


{% block h1 %}Tags{% endblock %}


{% block conteudo %}


<ul>
    {% for tag in lista %}
    <li><a href="{{ tag.get_absolute_url }}">{{ tag }}</a></li>
    {% endfor %}
</ul>


{% endblock conteudo %}
```

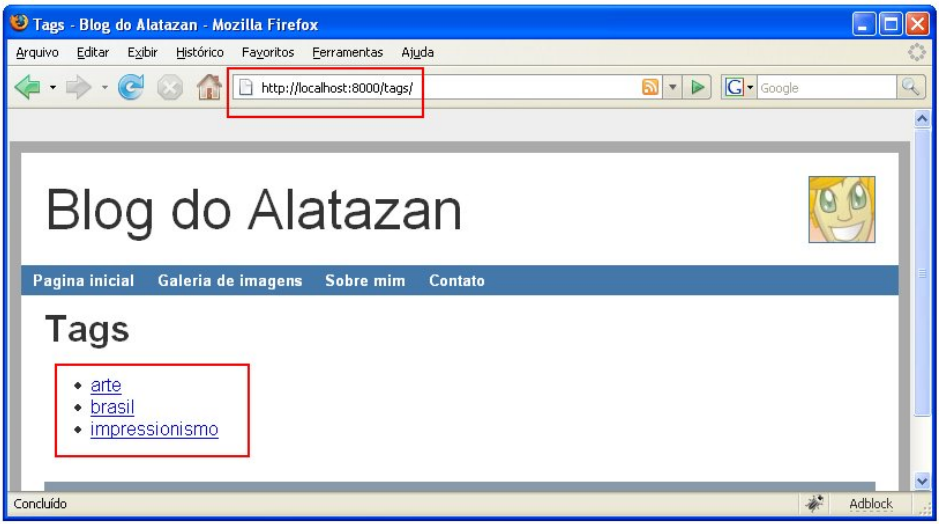
Temos aqui um template muito semelhante ao da **lista de álbuns** que trabalhamos no capítulo anterior.

Salve o arquivo. Feche o arquivo.

Agora vá ao navegador e carregue a seguinte URL:

```
http://localhost:8000/tags/
```

Veja o que aparece:



Impressionado?

Agora que tal criar...

A página da tag e seus objetos!

Abra novamente o arquivo **"urls.py"** da aplicação **"tags"** para edição e acrescente a seguinte URL:

```
    url(r'^(?P<tag_nome>.*?)/$', 'tag', name='tag'),
```

E o arquivo vai ficar assim:

```
from django.conf.urls.defaults import *


urlpatterns = patterns('tags.views',
    url(r'^$', 'tags', name='tags'),
    url(r'^(?P<tag_nome>.*?)/$', 'tag', name='tag'),
)
```

Salve o arquivo. Feche o arquivo.

Agora, abra o arquivo **"views.py"** da mesma pasta para edição, e acrescente as seguintes linhas de código ao final:

```
def tag(request, tag_nome):
    tag = get_object_or_404(Tag, nome=tag_nome)
    return render_to_response(
        'tags/tag.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

É apenas uma *view* básica que carrega a **tag** e faz uso do template **"tags/tag.html"**, mas ela precisa também da função **"get_object_or_404()"**. Para ajustar isso, localize esta linha:

```
from django.shortcuts import render_to_response
```

E acrescente abaixo a seguinte linha:

```
from django.shortcuts import get_object_or_404
```

Salve o arquivo. Feche o arquivo.

Agora, vá até a pasta **"templates"** da pasta da aplicação, e abra a pasta **"tags"**, criando o arquivo **"tag.html"** com o seguinte código dentro:

```
{% extends "base.html" %}


{% block titulo %}Tag "{{ tag.nome }}" -
{{ block.super }}{% endblock %}


{% block h1 %}Tag "{{ tag.nome }}"{% endblock %}


{% block conteudo %}


<ul>
{% for item in tag.tagitem_set.all %}
<li>
<a href="{{ item.objeto.get_absolute_url }}">{{ item.objeto }}
</a>
</li>
{% endfor %}
```

```
</ul>


{% endblock conteudo %}
```

Bom, agora veja bem: temos uma novidade aqui:

```
    {% for item in tag.tagitem_set.all %}
```

Essa linha faz referência a uma lista chamada **"tag.tagitem_set.all"** que não temos a menor ideia de onde veio.

Acontece que quando criamos o relacionamento entre as classes de modelo **"Tag"** e **"TagItem"** lá no começo do capítulo, a existência do campo **"tag"** como uma **"ForeignKey"** da classe **"Tag"** em **"TagItem"** faz o Django criar uma referência recíproca, fazendo o caminho inverso.

Ou seja: se a classe TagItem **faz parte** de uma Tag, então a Tag **possui** um **conjunto** de TagItem, e isso é representado por:

```
tag.tagitem_set
```

E quando acrescentamos mais o elemento **all** estamos fazendo referência ao método **".all()"**, que retorna todos os objetos:

```
tag.tagitem_set.all()
```

Só que no template, métodos devem ser chamados **sem os parênteses**. É apenas uma questão de sintaxe:

```
{{ tag.tagitem_set.all }}
```

Por isso que fizemos um laço nele, usando a template tag **{% for %}**.

Agora, partindo para a novidade seguinte, veja esta outra linha:

```
<li>
<a href="{{ item.objeto.get_absolute_url }}">{{ item.objeto }}
</a>
</li>
```

Ali nós fizemos uma referência ao atributo **"objeto"** do item. Mas esse atributo não existe. Então, precisamos **fazê-lo existir**!

Salve o arquivo. Feche o arquivo.

Conhecendo Generic Relations

Abra agora o arquivo **"models.py"** da aplicação **"tags"** e localize a seguinte linha:

```
    object_id = models.PositiveIntegerField(db_index=True)
```

Acrescente esta linha de código abaixo dela:

```
    objeto = GenericForeignKey('content_type', 'object_id')
```

O que esta nova linha de código faz é declarar uma **Generic Relation**, ou seja, nós estamos carregando o objeto relacionado (seja ele um **Artigo** ou uma **Imagem**) no atributo **"objeto"**. Para isso, usamos a classe **"GenericForeignKey"**.

Agora localize esta linha:

```
from django.contrib.contenttypes.models import ContentType
```

E acrescente estas abaixo dela:

```
from django.contrib.contenttypes.generic import GenericForeignKey
from django.core.urlresolvers import reverse
```

Ali nós importamos a classe, que está num pacote da aplicação **"contenttypes"** que oferece funcionalidades para **Generic Relations**.

E a segunda importação nada tem a ver com Generic Relations, mas vai ajudar a outro ajuste que vai dar vida à página de **tags**:

Localize a seguinte linha:

```
    nome = models.CharField(max_length=30, unique=True)
```

E acrescente este bloco de código abaixo dela:

```
    def get_absolute_url(self):
        return reverse('tag', kwargs={'tag_nome': self.nome})
```

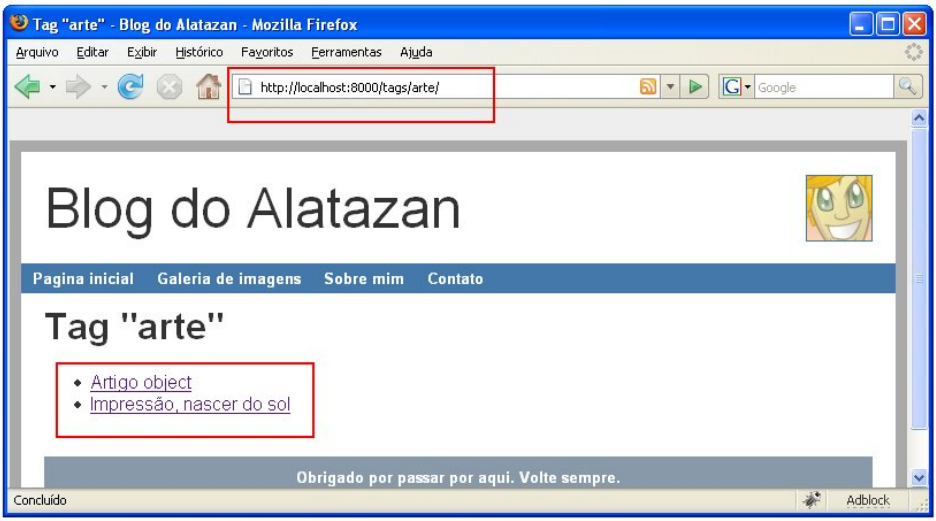
O arquivo **"models.py"** completo agora ficou assim:

```
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.generic import GenericForeignKey
from django.core.urlresolvers import reverse


class Tag(models.Model):
    nome = models.CharField(max_length=30, unique=True)


    def get_absolute_url(self):
        return reverse('tag', kwargs={'tag_nome': self.nome})


    def __unicode__(self):
        return self.nome
```

```
class TagItem(models.Model):
    class Meta:
        unique_together = ('tag', 'content_type', 'object_id')

    tag = models.ForeignKey('Tag')
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField(db_index=True)
    objeto = GenericForeignKey('content_type', 'object_id')
```

Salve o arquivo. Feche o arquivo.

Volte ao navegador, atualize a página com **F5** e clique sobre uma das tags listadas, veja como ela se sai:



Fascinante! É incrível como as **Generic Relations** trabalham!

Mas como você descobriu, o artigo ficou com um nome estranho... isso porque lá uns capítulos atrás, quando criamos a aplicação **"blog"**, nós nos esquecemos de fazer uma coisinha.

Ajustando a representação textual do artigo

Na pasta da aplicação **"blog"**, abra o arquivo **"models.py"** para edição e localize a seguinte linha:

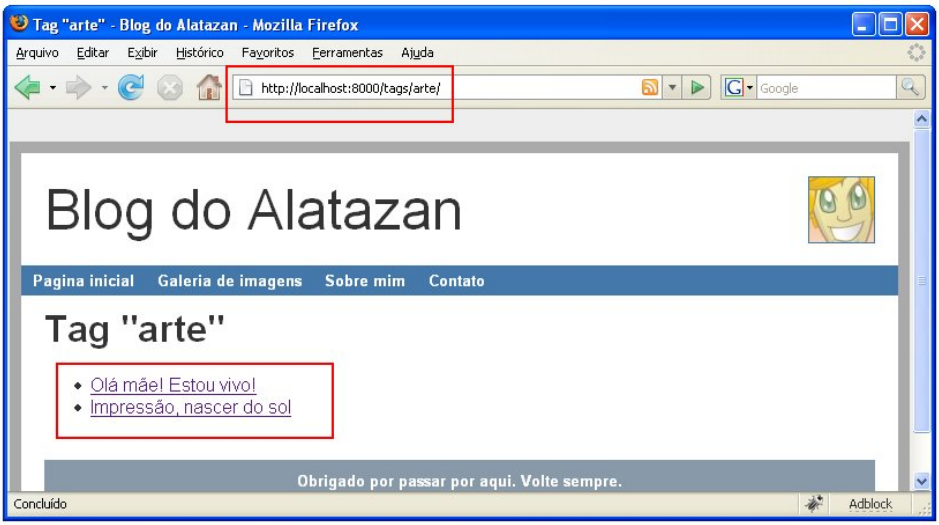
```
        return reverse(
        'blog.views.artigo', kwargs={'slug': self.slug}
        )
```

Agora acrescente este trecho de código logo abaixo dela:

```
    def __unicode__(self):
        return self.titulo
```

Salve o arquivo. Feche o arquivo.

Volte ao navegador, atualize com **F5** e veja como ficou:



Mostrando as tags do artigo com um Template Filter

Tudo isso é impressionante, mas ainda não acabou. Nós precisamos também ajustar o **template do artigo** para mostrar suas **tags**, e não há um caminho mais fácil de fazer algo *cool* no template do que criar um **Template Filter** pra isso!

Template Filters são funções usadas em templates para filtrar um objeto e retornar alguma outra coisa, e esta "outra coisa" pode ser filtrada por outro **Template Filter**, e assim por diante.

Na pasta da aplicação **"tags"**, crie uma nova pasta, chamada **"templatetags"**, e dentro dela, crie um arquivo vazio, chamado **"__init__.py"**. Crie ainda outro arquivo, chamado **"tags_tags.py"** com o seguinte código dentro:

```
from django.template import Library
```

```
from django.contrib.contenttypes.models import ContentType


from tags.models import TagItem


register = Library()


@register.filter
def tags_para_objeto(objeto):
    tipo_dinamico = ContentType.objects.get_for_model(objeto)


    itens = TagItem.objects.filter(
        content_type=tipo_dinamico,
        object_id=objeto.id,
        )


    return [item.tag for item in itens]
```

Vamos descobrir o que foi feito neste código?

Na primeira linha, importamos a classe **"Library"** para templates de Django. O que nós estamos fazendo agora é criar uma **biblioteca de utilidades para templates** e para isso precisamos da classe **"Library"**

Já na segunda linha, importamos a classe **"ContentType"**, que você já conhece bem, e é nossa companheira para tipos dinâmicos.

```
from django.template import Library
from django.contrib.contenttypes.models import ContentType
```

Em seguida importamos a classe de modelo **"TagItem"**, pois vamos precisar dela:

```
from tags.models import TagItem
```

Na linha a seguir, iniciamos a biblioteca de utilidades. É a ela que vamos incluir o nosso novo **template filter**:

```
register = Library()
```

E é no restante do código que fazemos isso. Primeiro registramos o **template filter** na biblioteca. Ele vai receber um **objeto**, carregar seu **tipo dinâmico**, carregar seus vínculos com **tags** e finalmente fazer um *list comprehension* para retornar a **lista das tags**:

```
@register.filter
```

```
def tags_para_objeto(objeto):
    tipo_dinamico = ContentType.objects.get_for_model(objeto)

    itens = TagItem.objects.filter(
        content_type=tipo_dinamico,
        object_id=objeto.id,
        )

    return [item.tag for item in itens]
```

Salve o arquivo. Feche o arquivo.

Agora abra para edição o arquivo de template **"artigo.html"** da pasta **"blog/templates/blog"**, partindo da pasta do projeto, e localize esta linha:

```
{% load comments %}
```

Modifique para ficar assim:

```
{% load comments tags_tags %}
```

Com isso, acrescentamos a biblioteca que acabamos de criar ao **template do artigo**. Agora localize esta outra linha:

```
{% block conteudo %}
```

E acrescente a seguinte linha abaixo dela:

```
<p>
Tags:
{% for tag in artigo|tags_para_objeto %}
<a href="{{ tag.get_absolute_url }}">{{ tag }}</a>
{% endfor %}
</p>
```

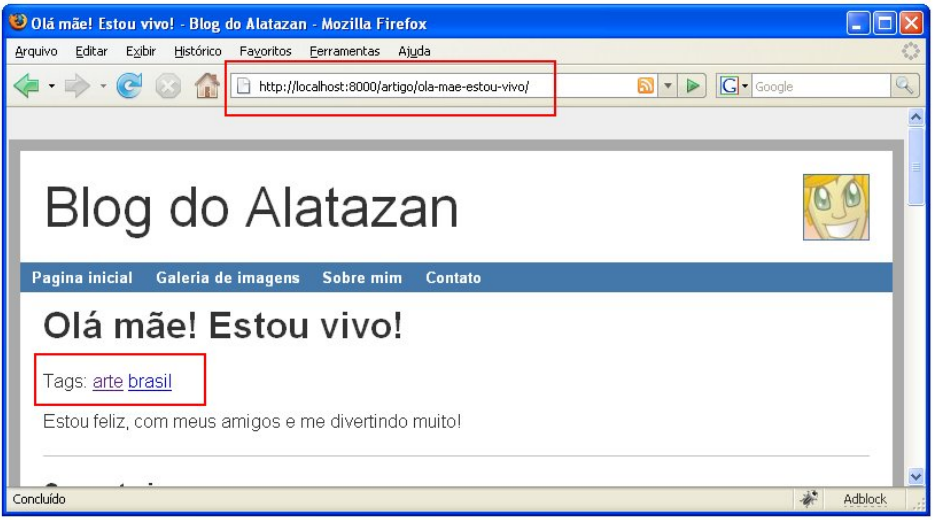
Observe que fazemos uso de nosso **template filter** dentro da tag **{% for tag in artigo|tags_para_objeto %}**.

Salve o arquivo. Feche o arquivo.

Agora vamos ao navegador para ver o efeito do que fizemos. Carregue a página de um artigo, assim:

```
http://localhost:8000/artigo/ola-mae-estou-vivo/
```

Pois agora este é o resultado que você vê no navegador:



Muito bacana esse tal **template filter** hein?

Mas agora nós vamos criar um template só para exibir as tags, e poder reusá-lo em outros templates com o mínimo de trabalho...

Volte ao arquivo de template **"artigo.html"** que acabamos de modificar e localize esta linha:

```
{% load comments tags_tags %}
```

Volte-a para como estava antes, modificando para ficar assim:

```
{% load comments %}
```

Agore localize este bloco de código e remova-o:

```
<p>
Tags:
{% for tag in artigo|tags_para_objeto %}
<a href="{{ tag.get_absolute_url }}">{{ tag }}</a>
{% endfor %}
</p>
```

No lugar do trecho de código que removeu, escreva este:

```
{% with artigo as objeto %}
{% include "tags/tags_para_objeto.html" %}
{% endwith %}
```

Esse trecho de código vai criar um apelido para o artigo numa variável chamada **"objeto"**, que será válida dentro do bloco da template tag **{% with %}**. E então o template **"tags/tags_para_objeto.html"** será incluído, para cumprir o mesmo papel que tinha agorinha mesmo.

O resultado final do arquivo **"artigo.html"** será este:

```
{% extends "base.html" %}

{% load comments %}

{% block titulo %}{{ artigo.titulo }} -
{{ block.super }}{% endblock %}

{% block h1 %}{{ artigo.titulo }}{% endblock %}

{% block conteudo %}
{% with artigo as objeto %}
{% include "tags/tags_para_objeto.html" %}
{% endwith %}

{{ artigo.conteudo }}

<div class="comentarios">
  <h3>Comentarios</h3>

  {% get_comment_list for artigo as comentarios %}
  {% for comentario in comentarios %}
  <div class="comentario">
    Nome: {{ comentario.name }}<br/>
    {% if comentario.url %}URL: {
    { comentario.url }
    }{% endif %}<br/>
    {{ comentario.comment|linebreaks }}
    <hr/>
  </div>
```

```
  {% endfor %}


  <h3>Envie um comentario</h3>


  {% render_comment_form for artigo %}


</div>
{% endblock %}
```

Salve o arquivo. Feche o arquivo.

Agora vá até a pasta **"tags/templates/tags"** partindo da pasta do projeto e crie um novo arquivo chamado **"tags_para_objeto.html"** com o seguinte código dentro

```
{% load tags_tags %}


<p>
Tags:
{% for tag in objeto|tags_para_objeto %}
<a href="{{ tag.get_absolute_url }}">{{ tag }}</a>
{% endfor %}
</p>
```

Observe que desta vez, esta linha se refere à variável **"objeto"** e não à variável **"artigo"**:

```
{% for tag in objeto|tags_para_objeto %}
```

Salve o arquivo. Feche o arquivo.

Agora que tal fazer o mesmo com a página de imagens?

Então abra para edição o arquivo **"imagem.html"** da pasta **"galeria/templates/galeria"**, partindo da pasta do projeto, e localize a seguinte linha:

```
    {{ imagem.descricao|linebreaks }}
```

Acrescente o seguinte trecho de código abaixo dela:

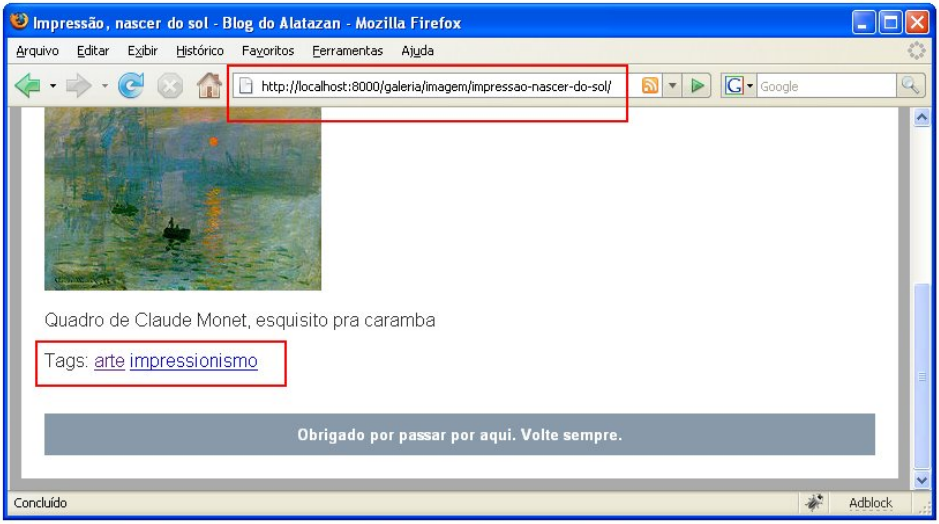
```
    {% with imagem as objeto %}
    {% include "tags/tags_para_objeto.html" %}
    {% endwith %}
```

Observe que a template tag **{% with imagem as objeto %}** faz referência à variável **"imagem"**, mas o apelido ainda é **"objeto"**.

Salve o arquivo. Feche o arquivo. Volte ao navegador, carregue a página de uma imagem, como esta URL, por exemplo:

```
http://localhost:8000/galeria/imagem/impressao-nascer-do-sol/
```

Veja o resultado:



Seja sincero com você mesmo: você conhece alguma ferramenta tão **elegante e produtiva** quanto esta?

Um item para as Tags no menu

Agora vamos criar um item no menu para **tags** em seu site? Vamos? Vamos?!

Abra o arquivo **"base.html"** da pasta **"templates"** do projeto para edição, e localize esta linha:

```
    <li>
    <a href="{% url galeria.views.albuns %}">Galeria de imagens
    </a>
    </li>
```

Acrescente a seguinte linha abaixo dela:

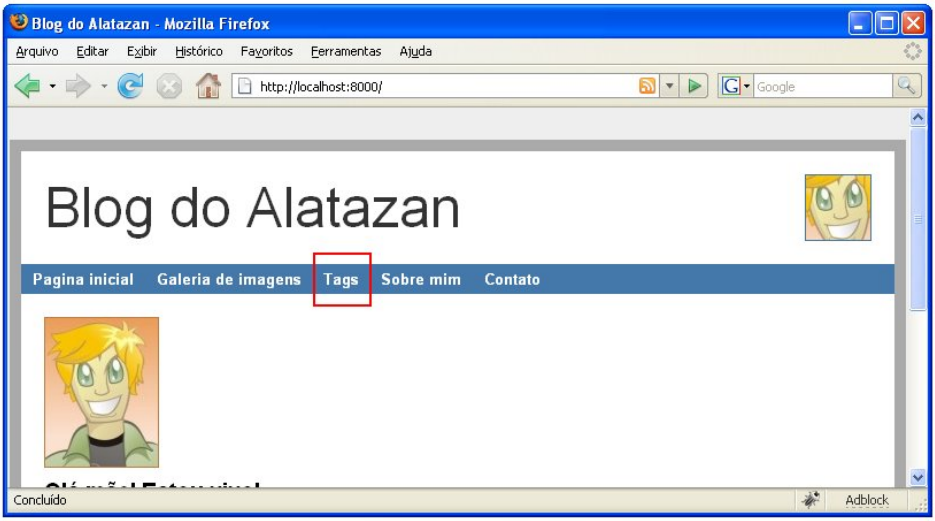
```
    <li><a href="{% url tags.views.tags %}">Tags</a></li>
```

Salve o arquivo. Feche o arquivo.

Agora novamente no navegador, carregue qualquer página do site, esta por exemplo:

```
http://localhost:8000/
```

Veja como ela ficou:



Pronto! Mais uma batalha vencida, e com muita elegância!

Tudo agora em outra língua...

Alatazan olhou pra fora pela janela e o Sol já não estava mais lá... o céu um pouco mais escuro fez ele perceber que o tempo havia passado incrivelmente rápido, e já era hora de ir embora, havia um cheirinho de chuva no ar...

Naquele dia pela manhã, a aula fora um pouco chata e comentaram sobre uma certa "oportunidade" para ingressar em um **Emprego Tradicional**.

Havia algo errado com a escola de Engenharia de Software, e Alatazan, Cartola e Nena eram parte de um grupo anônimo de rebeldes que não concordavam com algumas práticas. A camiseta preta por baixo de suas roupas era uma forma de se identificarem, e eles não eram muitos...

Tudo isso se passou de relance na memória de Alatazan, e ele não notou aqueles cabelos avermelhados chegarem à janela e uma voz doce comentar, com suas costas encostadas na beirada da janela, uma mão segurando um lápis enquanto a outra segurou seu ombro:

- Vamos embora?

Alatazan se assustou um pouco, pois estava um tanto absorto naqueles pensamentos...

- Hã? Bom... me deixa resumir a aventura de hoje e nós vamos...
    - Nós criamos uma nova aplicação, chamada **"tags"**;
    - A aplicação possui duas classes de modelo: uma para as tags, outra para o vínculo delas com os objetos;
    - Todo esse vínculo é feito usando **Generic Relations**, que faz uso de três elementos: um campo **"content_type"** para o tipo dinâmico, outro **"object_id"** para o **id** do objeto e um atributo **"objeto"** que é um **GenericForeignKey**;
    - Instalamos a aplicação no projeto e geramos as tabelas no banco de dados;
    - Modificamos o **Admin** das classes que relacionamos para ter um novo campo calculado, chamado **"tags"**, este campo salva e carrega as tags do objeto usando funções que criamos para isso;
    - Tudo é feito de forma discreta, com umas poucas mudanças no template e no **admin** das aplicações, sem interferências da camada de modelo ou no banco de dados;
    - Conhecemos um pouco de *list comprehensions* e do método **".join()"** para concatenar strings;
    - Criamos templates para a aplicação **"tags"**;
    - Criamos um **template filter** para mostrar as tags dos objetos em templates;
    - Criamos um template para fazer uso desse template filter e ser incluído nos templates dos objetos que tenham tags;
    - Acrescentamos um item ao menu para mostrar isso tudo.
- É isso **grande** *Alata*, dia de muitas novidades...
- Yep! Agora vamos fazer assim: eu quero ver aquela coisa da **internacionalização**, pode ser?
- Ô! Demorou!

Capítulo 22: O mesmo site em vários idiomas

O **Instituto de Previsões de Longuíssimo Prazo Mesmo** de Katara, uma instituição respeitadíssima na galáxia, tem representantes em diversos planetas e usa seu imenso banco de dados para indicar tendências **econômicas**, **políticas** e **meteorológicas** para toda parte.

As áreas de previsão econômica e política são um sucesso e em geral são confirmadas com no máximo **5 séculos**.

Com base em informações desse instituto, Alatazan escolheu seu destino no Planeta Terra e também aprendeu **Português, Inglês e Espanhol** usando um moderníssimo método dinâmico de aprendizado de idiomas, que se baseia na teoria de que a cada 17 mil anos todo o conjunto de expressões de uma língua se repetem, mesmo.

Como seu blog foi criado em **português**, ele tirou uma dessas últimas madrugadas que seus vizinhos fizeram festa pra refazer o site todo em **inglês e espanhol**, mas estava em dúvida de como faria pra deixar isso funcionando no site.

- Que zica é essa, Alatazan?
- Bom, é que eu quero que o site tenha três idiomas, e comecei a fazer outros templates pra adiantar o trabalho.
- Não cara, se a Nena estivesse aqui te daria um peteleco na cabeça...
- Mas o que há de errado aí? Eu até conferi no dicionário...
- Não, não... é que não é assim que se faz, vamos deletar isso tudo e começar do zero, você **complicou** as coisas...

Conhecendo o suporte à internacionalização

Os recursos de **internacionalização** do Django só não fazem cair o queixo de

dois tipos de pessoas: quem não a conhece e quem já se acostumou com ela.

Basicamente o que você precisa fazer é preparar o seu projeto com algumas práticas, e o restante o próprio Django faz, de forma transparente, sem precisar de URLs alternativas ou golpes de *ninja*.

Preparando as settings do projeto

Então vamos começar pelas **definições** do projeto.

Na pasta do projeto, abra o arquivo **"settings.py"** para edição e localize a seguinte linha:

```
LANGUAGE_CODE = 'pt-br'
```

Agora acrescente abaixo dela as seguintes linhas de código:

```
LANGUAGES = (
    ('pt-br', u'Português'),
    ('en', u'Inglês'),
    ('es', u'Espanhol'),
)
```

Isso foi feito porque queremos ter **3 idiomas** em nosso site: **Português**, **Inglês** e **Espanhol**.

Humm... mas tem um problema. Nos arquivos do Python, quando se quer informar qualquer caractere especial (como **"ê"** por exemplo) é preciso fazer uma coisinha: adicionar a seguinte linha ao início do arquivo:

```
# -*- coding: utf-8 -*-
```

Isso é necessário porque o Python trabalha em diversos idiomas, e a codificação padrão não permite caracteres estranhos ao idioma **inglês**. Quando se acrescenta a linha acima no arquivo, o Python entende que esse arquivo deve ser interpretado no formato **Unicode**, então todos os caracteres especiais passam a serem suportados.

Agora localize a seguinte linha de código:

```
    'django.contrib.auth.middleware.AuthenticationMiddleware',
```

Acrescente esta linha abaixo dela:

```
    'django.middleware.locale.LocaleMiddleware',
```

E a setting **"MIDDLEWARE_CLASSES"** passa a ficar assim:

```
MIDDLEWARE_CLASSES = (
'django.middleware.common.CommonMiddleware',
'django.contrib.sessions.middleware.SessionMiddleware',
```

```
'django.contrib.auth.middleware.AuthenticationMiddleware',
'django.middleware.locale.LocaleMiddleware',
'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',
)
```

O middleware **"django.middleware.locale.LocaleMiddleware"** é quem faz a mágica de saber qual é o idioma usado ou preferido do usuário, e repassar essa informação ao restante do Django. Mas calma, nós vamos chegar lá.

Salve o arquivo. Feche o arquivo.

Agora execute o projeto - caso ele ainda não foi executado - clicando duas vezes sobre o arquivo **"executar.bat"** da pasta do projeto.

Preparando os templates...

O próximo passo é ajustar os nossos templates para que sejam **internacionalizados**. Esse trabalho consiste em abrir cada um dos arquivos de template e fazer duas coisas:

1. Acrescentar a template tag **{% load i18n %}**;
2. Mudar as palavras que devem ser traduzidas para usarem a template tag **{% trans %}**

Então vamos começar pelo título do site, apenas ele...

Vá até a pasta **"templates"** da pasta do projeto e abra o arquivo **"base.html"** para edição. Localize a seguinte linha:

```
<title>{% block titulo %}Blog do Alatazan{% endblock %}</title>
```

Modifique esta linha, para ficar assim:

```
<title>
{% block titulo %}{% trans "Blog do Alatazan" %}{% endblock %}
</title>
```

Agora localize esta outra linha:

```
    Blog do Alatazan
```

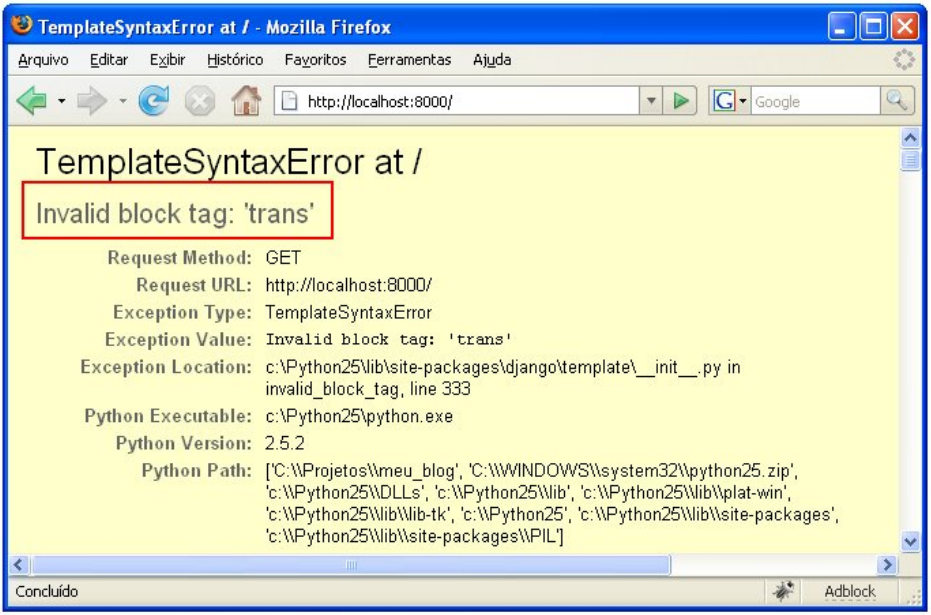
Modifique-a para ficar assim:

```
    {% trans "Blog do Alatazan" %}
```

Salve o arquivo e abra o navegador na seguinte URL:

```
http://localhost:8000/
```

Veja o que acontece:



Temos uma mensagem de erro:

```
TemplateSyntaxError at /

Invalid block tag: 'trans'
```

E a mensagem de erro significa o que você deve estar pensando: a template tag **{% trans %}** é inválida. Não existe.

A template tag **{% trans %}** tem o papel de declarar um termo para que ele seja traduzido para o idioma escolhido pelo usuário. Portanto a seguinte expressão...

```
{% trans "Blog do Alatazan" %}
```

...define que quando o idioma escolhido for o **inglês**, será exibido ali algo como **"Alatazan's Blog"**. Quando for **espanhol** será exibido **"Diário del Alatazan"** ou algo parecido, e assim por diante.

Mas essa template tag faz parte de uma **biblioteca para templates** que não está presente no sistema de templates padrão. Então é preciso carregar essa biblioteca. Para isso, acrescente a seguinte linha ao início do arquivo:

```
{% load i18n %}
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, pressione **F5** para atualizar e você pode ver que tudo voltou ao normal.

Instalando Gettext no Windows

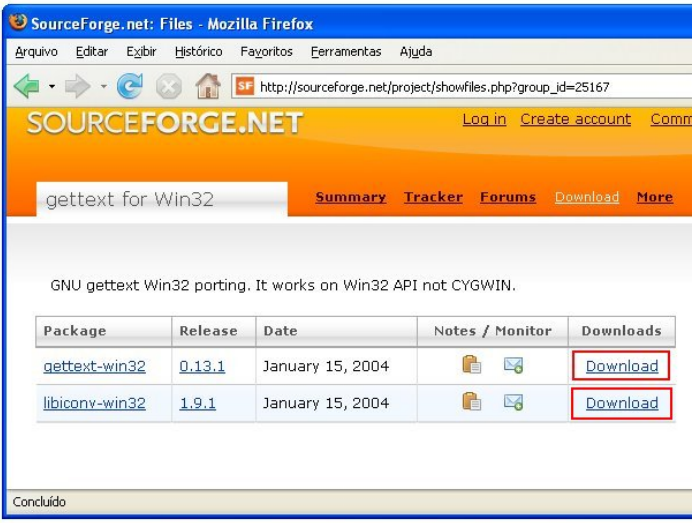
Agora que temos um termo para ser traduzido (**"Blog do Alatazan"**), precisamos escrever as traduções desse termo. Mas isso é feito em um lugar separado.

Quem trabalha com o **Windows** possui uma dificuldade a mais aqui. Porque na verdade o suporte à internacionalização do Django é baseado no **GNU Gettext** e depende dele. Este software é uma biblioteca fantástica que é **software livre** e é usada massivamente no Linux, Unix e outros sistemas operacionais. No Windows, ela deve ser instalada e o processo não é dos mais amigáveis, mas é bastante simples.

Primeiro, você deve ir até esta URL:

```
http://sourceforge.net/projects/gettext
```

Clique na caixa **"Download"** e a seguinte página será mostrada:



Você deve clicar no primeiro link destacado e localizar os seguintes arquivos para download:

- gettext-runtime-0.13.1.bin.woe32.zip
- gettext-tools-0.13.1.bin.woe32.zip

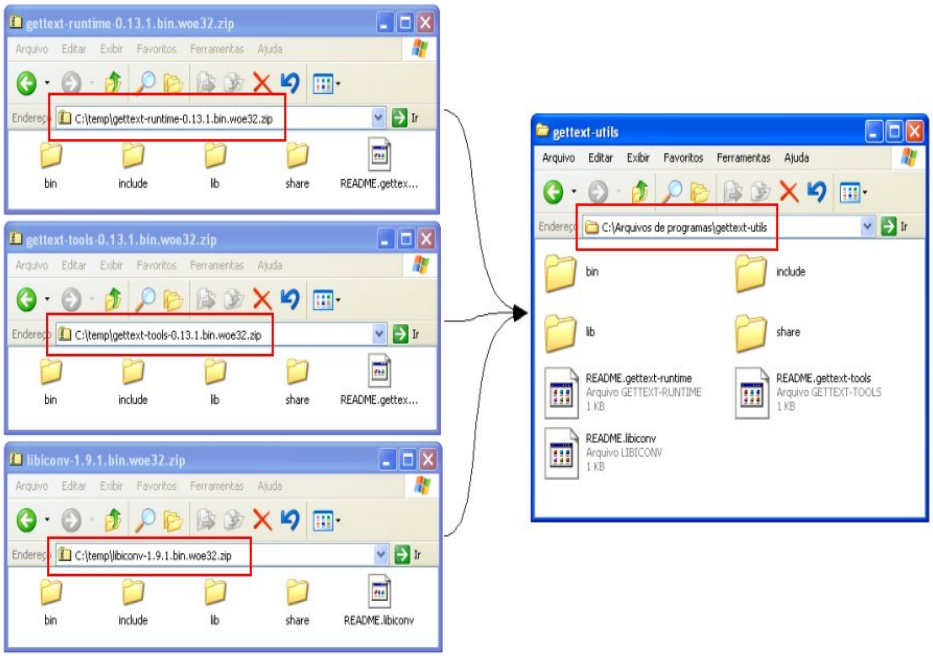
Feito o download, volte à página anterior e clique no segundo link destacado. Localize o seguinte arquivo e faça o download:

- libiconv-1.9.1.bin.woe32.zip

Agora vá até a pasta de aplicativos instalados do HD (**c:\Arquivos de Programas**) e crie uma pasta chamada **"gettext-utils"**, para ficar assim:

```
c:\Arquivos de Programas\gettext-utils
```

Ao concluir os downloads, você deve extrair os 3 arquivos para a nova pasta (todos para a mesma pasta). Os 3 arquivos possuem pastas como **"bin"**, **"lib"** e outros, mas não se preocupe, eles serão fundidos e misturados uns aos outros mesmo.



Agora o próximo passo é adicionar a nova pasta à variável de ambiente **PATH** do Windows. Para isso faça o seguinte:

1. Pressione as teclas **Windows** + **Pause** (a tecla Windows é a tecla do **logotipo** do Windows).

2. Na janela aberta, clique na aba **"Avançado"** e depois disso, no botão **"Variáveis de ambiente"**.

3. Na caixa de seleção **"Variáveis do Sistema"** clique duas vezes sobre o item **"Path"**.

4. E ao final do campo **"Valor da Variável"**, adicione um ponto-e-vírgula e o caminho da nova pasta onde instalamos o **gettext** mais **"\bin"** ao final, assim:

```
;c:\Arquivos de Programas\gettext-utils\bin
```

Depois disso, clique nos botões **"Ok"** até voltar para o estado anterior.

Pronto! Agora com o **Gettext** instalado, podemos seguir em frente.

Usando makemessages para gerar arquivos de tradução

Agora voltando à pasta do projeto, crie um novo arquivo, chamado **"makemessages.bat"**, com o seguinte código dentro:

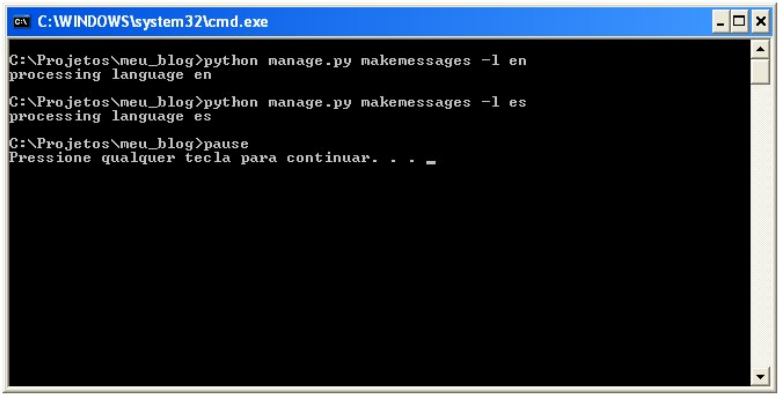
```
python manage.py makemessages -l en

python manage.py makemessages -l es

pause
```

Esses comandos vão criar os arquivos de tradução para os idiomas **"en"** (Inglês) e **"es"** (Espanhol).

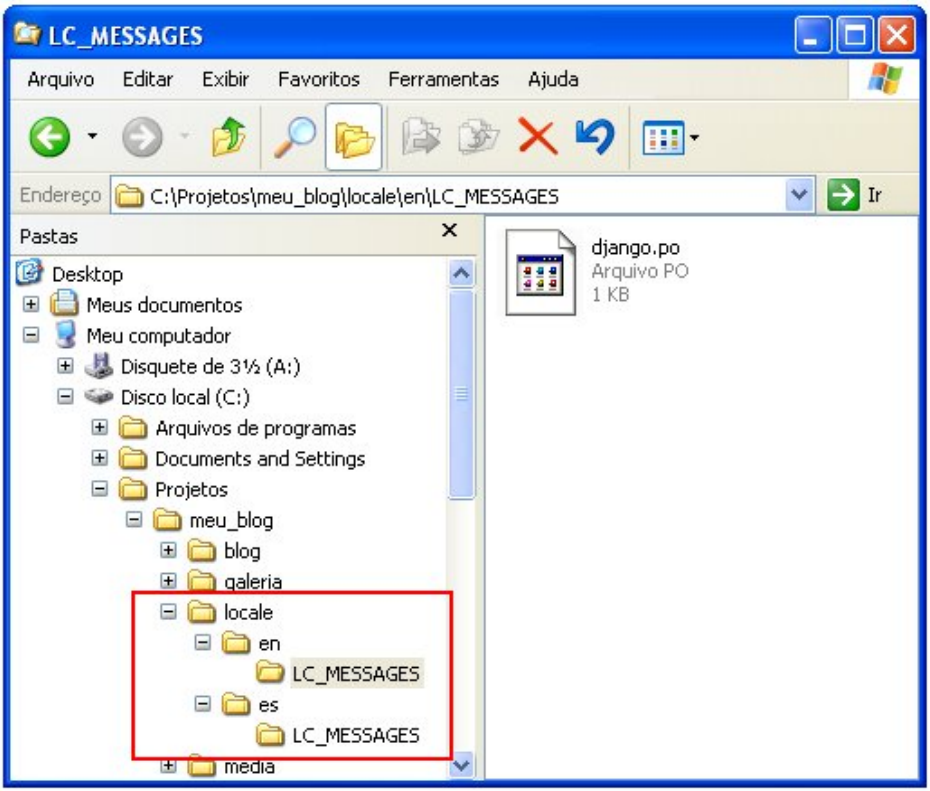
Salve o arquivo. Feche o arquivo.

Agora, ainda na pasta do projeto, precisamos criar a pasta onde esses arquivos serão gerados. Portanto crie uma nova pasta chamada **"locale"**.

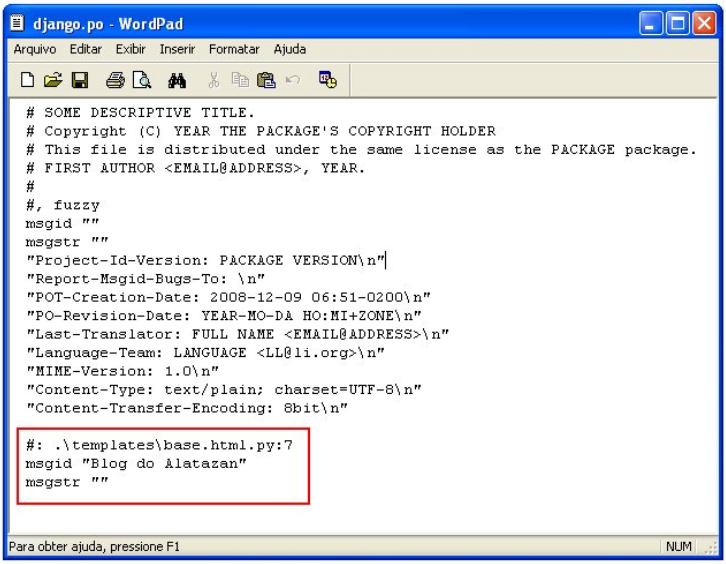
E por fim, para gerar os arquivos de tradução, clique duas vezes sobre o arquivo **"makemessages.bat"**. Isso vai fazer com que o Django localize todas as strings marcadas para tradução (o que inclui a nossa **{% trans "Blog do Alatazan" %}**) e colocar nesses arquivos. Veja:



E veja como os arquivos foram criados pelo **"makemessages"**:



Agora você está descobrindo que o Django criou as pastas **"en"** e **"es"** com seus respectivos arquivos, e nós vamos editá-los para fazer a tradução dos termos que precisamos. Portanto, na pasta **"locale/en/LC_MESSAGES"**, abra o arquivo **"django.po"** para edição, usando o **WordPad**, assim:



A escolha pelo **WordPad** é pelo seguinte motivo: o Django é construído e usado principalmente por pessoas que usam e trabalham com **Linux ou MacOS X**. A forma como o Django gera arquivos segue os padrões desses dois sistemas operacionais em muitos aspectos, e este arquivo de traduções é um deles. Caso você abra usando o **Bloco de Notas**, o arquivo será todo exibido de uma forma distorcida, sem saltos de linha, e o WordPad é o editor compatível com esse formato que o Windows dispõe. Por isso fizemos escolha por ele.

Agora localize este trecho de código:

```
#: .\templates\base.html.py:7
msgid "Blog do Alatazan"
msgstr ""
```

Modifique para ficar assim:

```
#: .\templates\base.html.py:7
msgid "Blog do Alatazan"
msgstr "Alatazan's Blog"
```

Observe que alteramos somente o conteúdo do atributo **"msgstr"** e mantivemos o restante do bloco inalterado. Isso é importante, pois o **Gettext** localiza suas traduções com base nos termos identificados pelo atributo **"msgid"**.

Salve o arquivo. Feche o arquivo.

Agora vamos traduzir o mesmo termo para o idioma **espanhol**. Vá até a pasta

"locale/es/LC_MESSAGES" e abra o arquivo **"django.po"** e localize o mesmo bloco de código:

```
#: .\templates\base.html.py:7
msgid "Blog do Alatazan"
msgstr ""
```

Modifique para ficar assim:

```
#: .\templates\base.html.py:7
msgid "Blog do Alatazan"
msgstr "Diario del Alatazan"
```

Da mesma forma, modificamos somente o atributo **"msgstr"**.

Salve o arquivo. Feche o arquivo.

Compilando os arquivos de tradução

Bom, não é só isso. Por questões de performance e outros motivos, é necessário compilar esses arquivos de extensão **.po** para gerar outros de extensão **.mo**, que serão de fato usados pelo Django.

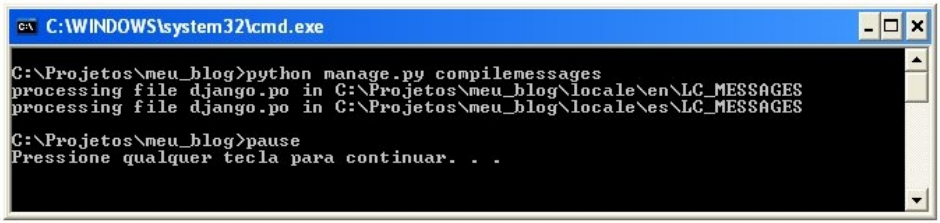
Para isso, vá até a pasta do projeto e crie um novo arquivo chamado **"compilemessages.bat"**, com o seguinte código dentro:

```
python manage.py compilemessages
pause
```

Esse comando vai compilar os arquivos e gerar outros com extensão **.mo** para cada um dos **.po**.

Salve o arquivo. Feche o arquivo.

Agora clique duas vezes sobre ele veja o que acontece:



Como as mensagens indicam, os arquivos foram compilados com sucesso. Compilar os arquivos usando este comando é necessário para **todas as vezes** que os arquivos de tradução forem modificados.

Mas isso não é suficiente, pois a página permanece inalterada, e em **português**.

Deixando que o usuário escolha seu idioma preferido

Pois agora vamos permitir que o usuário escolha o idioma do site como ele preferir. Para isso, vá até a pasta **"template"** da pasta do projeto e abra o arquivo **"base.html"**. Localize a seguinte linha:

```
<div id="menu">
```

Acima dela, acrescente esta outra:

```
    {% include "idiomas.html" %}
```

Salve o arquivo. Feche o arquivo.

Agora na mesma pasta, crie um novo arquivo chamado **"idiomas.html"** com o seguinte código dentro:

```
{% load i18n %}


{% get_available_languages as LANGUAGES %}


<div class="idiomas">
  <form action="/i18n/setlang/" method="post">


    <select name="language">
      {% for lang in LANGUAGES %}
      <option value="{{ lang.0 }}">{{ lang.1 }}</option>
      {% endfor %}
    </select>


    <input type="submit" value="{% trans "Mudar idioma" %}" />
  </form>
</div>
```

Bom, o arquivo não é grande, mas precisamos esclarecer parte por parte...

Aqui nós carregamos a **biblioteca de internacionalização para templates**, com a qual já fizemos contato agora há pouco:

```
{% load i18n %}
```

Agora nós carregamos todos os idiomas **disponíveis** no projeto. Você se lembra que nós declaramos a setting **"LANGUAGES"** lá no começo do capítulo? Pois

elas serão usadas aqui agora:

```
{% get_available_languages as LANGUAGES %}
```

O próximo passo é definir uma caixa e um **formulário manual** para o usuário selecionar seu idioma desejado:

```
<div class="idiomas">
  <form action="/i18n/setlang/" method="post">
```

A seguir, temos o **campo de seleção** que vai dar ao usuário as opções de escolha do idioma. O nome desse campo é **"language"** e suas opções serão os três idiomas que determinamos na setting **"LANGUAGES"**:

```
    <select name="language">
      {% for lang in LANGUAGES %}
      <option value="{{ lang.0 }}">{{ lang.1 }}</option>
      {% endfor %}
    </select>
```

Aqui é preciso fazer uma observação. Veja as variáveis **{{ lang.0 }}** e **{{ lang.1 }}**.

No sistema de templates do Django não é permitido referenciar aos itens de uma lista da forma como fazemos no Python (observe as linhas com **uma_lista[0]** e **uma_lista[1]**:

```
>>> uma_lista = ['sao paulo','rio','bh']
>>> uma_lista
['sao paulo', 'rio', 'bh']
>>> uma_lista[0]
'sao paulo'
>>> uma_lista[1]
'rio'
```

Então, nas templates, a sintaxe muda e as chaves (**[ ]**) devem ser trocadas por um ponto **antes do número do item**, assim:

```
{{ uma_lista.0 }}
{{ uma_lista.1 }}
```

Voltando à settings **"LANGUAGES"**, veja como ela foi declarada no arquivo **"settings.py"**:

```
LANGUAGES = (
    ('pt-br', u'Português'),
```

```
    ('en', u'Inglês'),
    ('es', u'Espanhol'),
)
```

Se para cada volta do laço da template tag **{% for lang in LANGUAGES %}** a variável **"lang"** representa uma linha da setting **"LANGUAGES"**, então no primeiro item da setting **{{ lang.0 }}** contém **'pt-br'** e **{{ lang.1 }}** contém **u'Português'**. No segundo item, elas são **'en'** e **u'Inglês'** respectivamente, e assim por diante.

Por fim, temos aqui abaixo o **botão de envio** da escolha feita pelo usuário:

```
    <input type="submit" value="{% trans "Mudar idioma" %}" />
```

Salve o arquivo. Feche o arquivo.

Mas se fizemos referência à URL **/i18n/setlang/**, então temos aqui uma nova tarefa, certo?

Declarando a URL de utilidades de internacionalização

Vá até a pasta do projeto, abra o arquivo **"urls.py"** para edição e acrescente a seguinte URL:

```
    (r'^i18n/', include('django.conf.urls.i18n')),
```

Agora o arquivo **"urls.py"** completo ficou assim:

```
from django.conf.urls.defaults import *
from django.conf import settings


# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()


from blog.models import Artigo
from blog.feeds import UltimosArtigos


urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index',
        {'queryset': Artigo.objects.all(), 'date_field':
'publicacao'}),
    (r'^admin/(.*)', admin.site.root),
    (r'^rss/(?P<url>.*)/$',
```

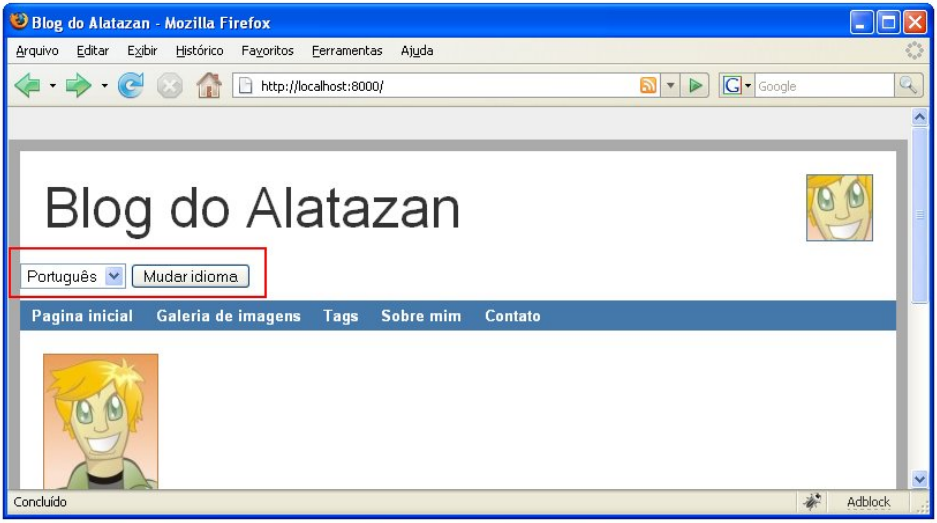
```
'django.contrib.syndication.views.feed',
        {'feed_dict': {'ultimos': UltimosArtigos}}),
    (r'^artigo/(?P<slug>[\w_-]+)/$', 'blog.views.artigo'),
    (r'^contato/$', 'views.contato'),
    (r'^comments/', include('django.contrib.comments.urls')),
    (r'^galeria/', include('galeria.urls')),
    (r'^tags/', include('tags.urls')),
    (r'^i18n/', include('django.conf.urls.i18n')),
)

if settings.LOCAL:
    urlpatterns += patterns('',
        (r'^media/(.*)$', 'django.views.static.serve',
         {'document_root': settings.MEDIA_ROOT}),
    )
```

Salve o arquivo. Feche o arquivo.

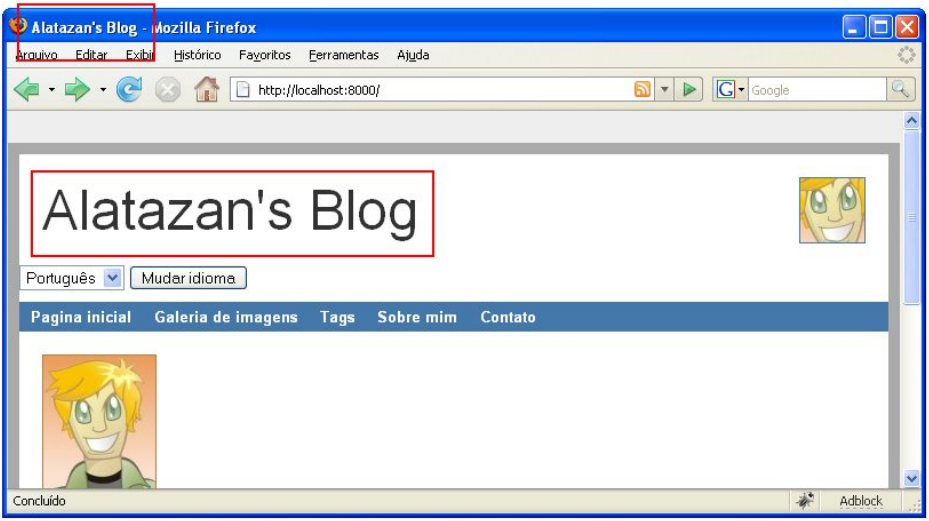
Agora volte ao navegador e carregue a URL principal do projeto:

```
http://localhost:8000/
```

Veja como a página é carregada:



Opa! Agora temos um formulário ali, para a escolha do idioma! Escolha o idioma **"Inglês"**, clique sobre o botão e veja como se sai:

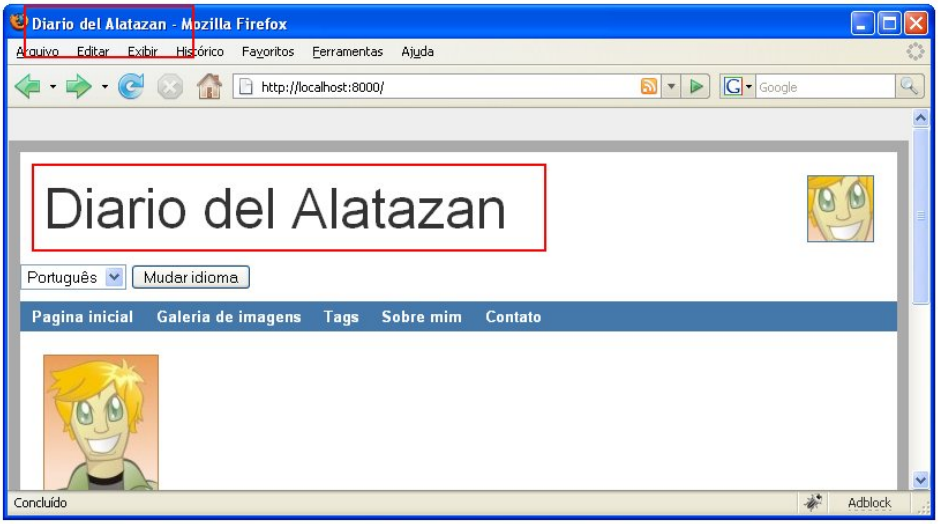


Pronto! Estamos navegando em inglês!

Uma vez que um idioma é selecionado, a opção escolhida é **armazenada na sessão**, e o idioma será trocado novamente somente quando a sessão expirar ou quando o usuário o fizer da mesma forma que você fez agora.

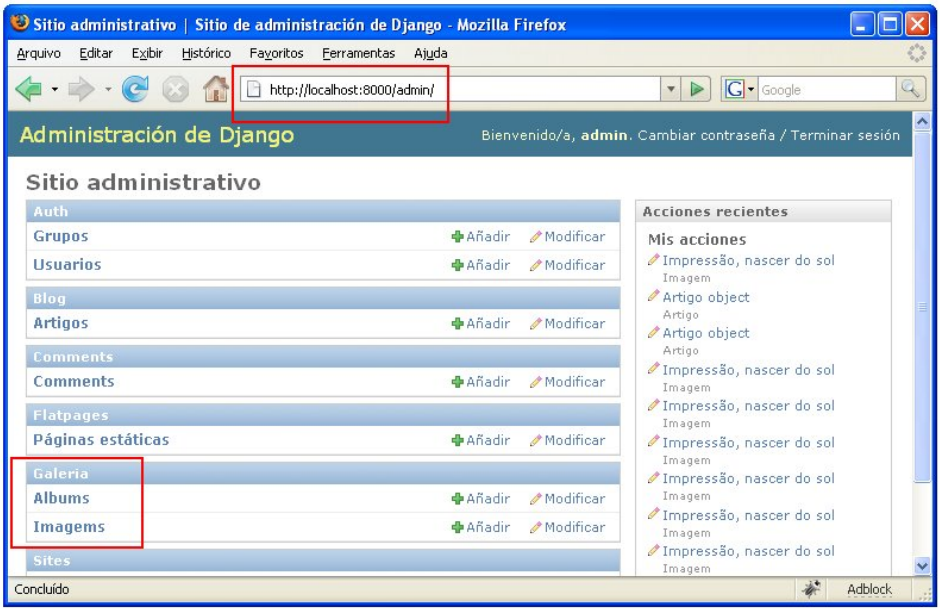
Além disso, quando o usuário entra pela primeira vez no site e enquanto não fez nenhuma escolha, o Django tenta se adequar ao idioma do navegador. Portanto, caso um usuário acesse seu site partindo de um navegador em espanhol, ele estará **automaticamente** navegando **em espanhol**!

Agora troque novamente o idioma, mas desta vez para **espanhol**. Veja:



Arriba muchacho! Estamos hablando em español!

Agora apenas carregue outra URL... a do **Admin**, por exemplo:



Mas você notou que o quadro destacado da aplicação **"Galeria"** tem uma palavra que não está em **espanhol**?

Usando a função de tradução em código Python: ugettext_lazy()

De acordo com o tradutor do Google, a palavra **"Imagems"** deveria ser **"Cuadros"**, isso sem falar que **"Imagems"** já está errada mesmo em português!

Então, vamos fazer assim: vá até a pasta da aplicação **"galeria"**, abra o arquivo **"models.py"** para edição e localize o seguinte bloco de código:

```
    class Meta:
        ordering = ('album','titulo',)
```

Acrescente estas novas linhas dentro dele:

```
        verbose_name = ugettext_lazy('Imagem')
        verbose_name_plural = ugettext_lazy('Imagens')
```

Observe que a primeira linha que incluímos aponta um **nome amigável** para essa classe de modelo, **no singular**, usando o atributo **"verbose_name"**. Mas além disso, ele também atribui a palavra **"Imagem"** dentro da função **"ugettext_lazy()"**, que equivale à template tag **{% trans %}**, mas para ser usada em código Python. Ou seja, estamos dizendo aí que a palavra **"Imagem"** deve ser traduzida:

```
        verbose_name = ugettext_lazy('Imagem')
```

A segunda linha tem o mesmo papel, mas para o caso de **nome amigável no plural**:

```
        verbose_name_plural = ugettext_lazy('Imagens')
```

Agora aquele bloco que localizamos e modificamos está assim:

```
    class Meta:
        ordering = ('album','titulo',)
        verbose_name = ugettext_lazy('Imagem')
        verbose_name_plural = ugettext_lazy('Imagens')
```

Mas como estamos usando a função **"ugettext_lazy()"**, devemos importá-la de algum lugar, certo? Pois então vá até o ínicio do arquivo e localize esta linha:

```
from django.core.urlresolvers import reverse
```

Agora acrescente esta outra abaixo dela:

```
from django.utils.translation import ugettext_lazy
```

Salve o arquivo. Feche o arquivo. Agora precisamos gerar novamente os arquivos de tradução, usando o utilitário **"makemessages.bat"** da pasta do projeto.

Depois que executar o **"makemessages.bat"**, vá até a pasta **"locale/en/LC_MESSAGES"**, partindo da pasta do projeto, e abra o arquivo **"django.po"**. Veja que agora ele já tem mais coisas:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE
# package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2008-12-09 18:49-0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: .\galeria\models.py:28
msgid "Imagem"
msgstr ""

#: .\galeria\models.py:29
msgid "Imagens"
msgstr ""

#: .\templates\base.html.py:7 .\templates\base.html.py:33
msgid "Blog do Alatazan"
msgstr "Alatazan's Blog"

#: .\templates\idiomas.html.py:14
msgid "Mudar idioma"
msgstr ""
```

Faça a tradução dos termos nos atributos **"msgstr"**, para ficar assim:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE
# package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2008-12-09 18:49-0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: .\galeria\models.py:28
msgid "Imagem"
msgstr "Picture"

#: .\galeria\models.py:29
msgid "Imagens"
msgstr "Pictures"

#: .\templates\base.html.py:7 .\templates\base.html.py:33
msgid "Blog do Alatazan"
msgstr "Alatazan's Blog"

#: .\templates\idiomas.html.py:14
msgid "Mudar idioma"
```

```
msgstr "Change language"
```

Salve o arquivo. Feche o arquivo.

Agora faça o mesmo com o arquivo **"django.po"** da pasta **"locale/es/LC_MESSAGES"**:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE
# package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2008-12-09 18:49-0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: .\galeria\models.py:28
msgid "Imagem"
msgstr ""

#: .\galeria\models.py:29
msgid "Imagens"
msgstr ""

#: .\templates\base.html.py:7 .\templates\base.html.py:33
msgid "Blog do Alatazan"
msgstr "Diario del Alatazan"
```

```
#: .\templates\idiomas.html.py:14
msgid "Mudar idioma"
msgstr ""
```

Modifique traduzindo os termos para ficar assim:

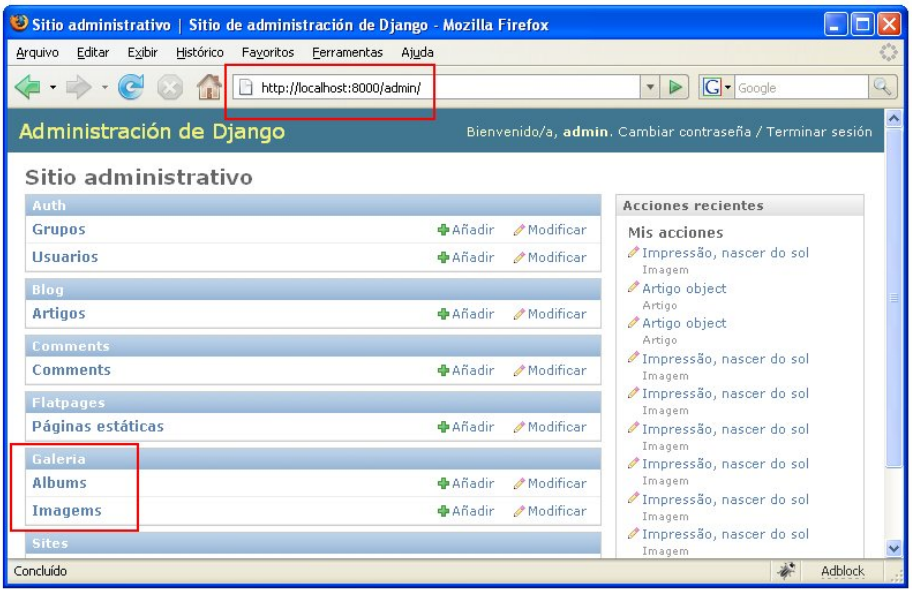
```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE
# package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2008-12-09 18:49-0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: .\galeria\models.py:28
msgid "Imagem"
msgstr "Cuadro"

#: .\galeria\models.py:29
msgid "Imagens"
msgstr "Cuadros"

#: .\templates\base.html.py:7 .\templates\base.html.py:33
msgid "Blog do Alatazan"
```

```
msgstr "Diario del Alatazan"


#: .\templates\idiomas.html.py:14
msgid "Mudar idioma"
msgstr "Cambiar de idioma"
```

Salve o arquivo. Feche o arquivo.

Agora é preciso **compilar** os arquivos de tradução novamente. Faça isso clicando duas vezes sobre o arquivo **"compilemessages.bat"** da pasta do projeto. Acompanhe as mensagens de erro para se certificar de que tudo ocorreu corretamente.

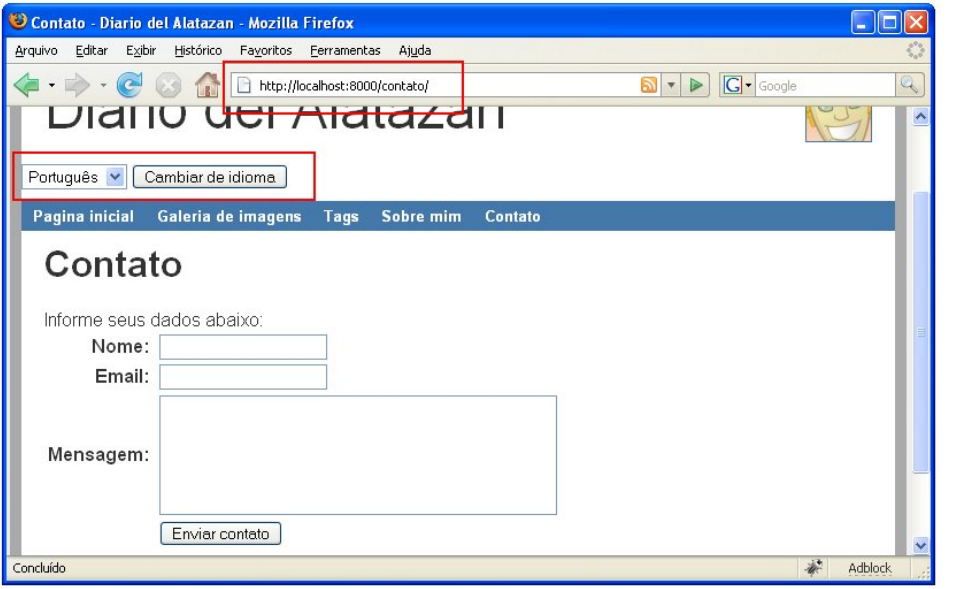
Agora volte ao navegador, na URL do Admin, atualize com **F5** e veja como ficou:



E voltando a navegar no site na URL de **Contato**:

```
http://localhost:8000/contato/
```

Veja como se sai:



Temos o **"Cambiar de idioma"** funcionando perfeitamente! Mas como vemos, nosso **formulário de contato** também deveria ser traduzido. Pois vamos fazer isso!

Usando a função ugettext()

Vá até a pasta do projeto, abra para edição o arquivo **"views.py"** e localize o seguinte bloco de código:

```
class FormContato(forms.Form):
    nome = forms.CharField(max_length=50)
    email = forms.EmailField(required=False)
    mensagem = forms.Field(widget=forms.Textarea)
```

Ele deve ser ajustado para ficar assim:

```
class FormContato(forms.Form):
    nome = forms.CharField(max_length=50, label=ugettext('Nome'))
    email = forms.EmailField(
    required=False, label=ugettext('E-mail')
    )
    mensagem = forms.Field(
    widget=forms.Textarea, label=ugettext('Mensagem')
```

```
    )
```

Observe que acrescentamos a função **"ugettext()"** para cada um dos campos do formulário.

Agora acrescente a seguinte linha ao início do arquivo:

```
from django.utils.translation import ugettext
```

A questão agora é: porquê usamos a função **"ugettext_lazy()"** no arquivo **"models.py"** e agora usamos a função **"ugettext()"** no arquivo **"views.py"**?

A função **"ugettext_lazy()"** é **preguiçosa**. Isso significa que a tradução é feita somente quando ela é requisitada, o que é relativamente melhor para o caso de classes de modelo, pois elas são constantemente utilizadas sem que a tradução de um termo seja necessário de fato.

Por outro lado a função **"ugettext()"** traduz a string instantaneamente, o que é melhor para casos como o de formulários dinâmicos e *views*, pois eles não são usados de maneira tão constante quanto classes de modelo.

Mas para não deixar o desenvolvedor confuso, há uma truque que facilita muito toda essa mistura de coisas.

Usando _() ao invés de ugettext() ou ugettext_lazy()

Localize a linha que você acabou de inserir:

```
from django.utils.translation import ugettext
```

Modifique para ficar assim:

```
from django.utils.translation import ugettext as _
```

Quando se usa o operador **"as"** numa importação do Python, você está fazendo aquela importação e dando um apelido ao pacote ou elemento importado. Então neste caso a função **"ugettext()"** é importada, mas seu nome neste módulo do Python será apenas um sublinhado, assim: **"_()"**.

Agora vá até este bloco de código:

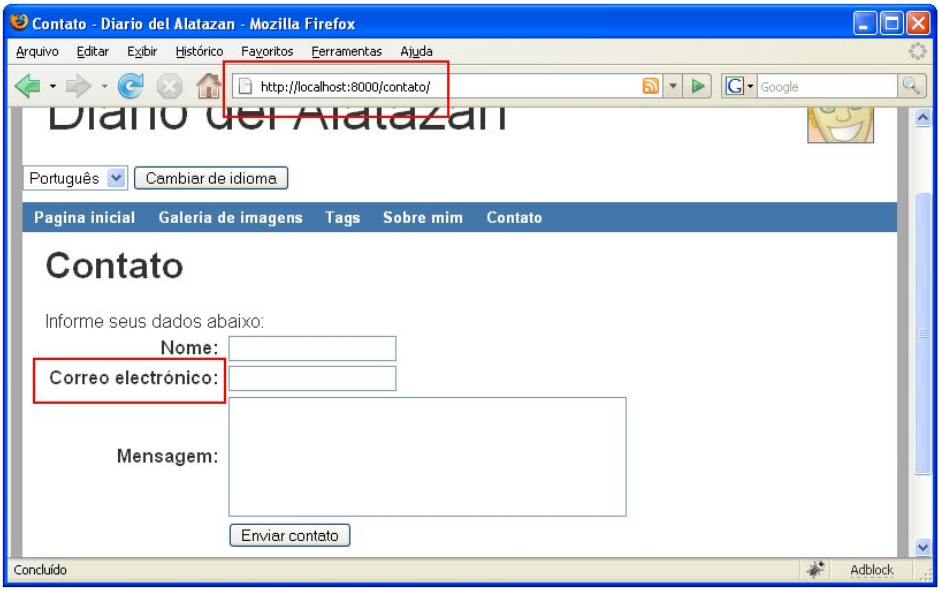
```
class FormContato(forms.Form):
    nome = forms.CharField(max_length=50, label=ugettext('Nome'))
    email = forms.EmailField(
    required=False, label=ugettext('E-mail')
    )
    mensagem = forms.Field(
    widget=forms.Textarea, label=ugettext('Mensagem')
    )
```

E troque as palavras **"ugettext"** por **"_"**, assim

```
class FormContato(forms.Form):
    nome = forms.CharField(max_length=50, label=_('Nome'))
    email = forms.EmailField(required=False, label=_('E-mail'))
    mensagem = forms.Field(
    widget=forms.Textarea, label=_('Mensagem')
    )
```

Esse truque faz com que seu código fique mais limpo e facilita uma eventual mudança de **"ugettext"** pela **"ugettext_lazy"**, quando for necessário.

Salve o arquivo. Feche o arquivo. Volte ao navegador e atualize a página com **F5**. Veja agora:



É isso aí moçada! Como você deve perceber, há muito mais termos para traduzir em nosso projeto. Que tal fazer o restante agora por conta própria?

Partindo para conhecer melhor o Admin

- É fantástico! É tão transparente e coeso que até assusta!
- Sim, meu caro, e veja só:
  - Primeiro você preparou as settings do projeto, declarando a setting

LANGUAGES e adicionando o middleware **"django.middleware.locale.LocaleMiddleware"**;

- Depois você ajustou alguns templates, usando a biblioteca **"i18n"** e a template tag **{% trans %}**;

- Aí você instalou o **Gnu Gettext** nessa coisa aí - isso não aconteceria no meu **MacOSX**;

- Criou dois arquivos de comandos para executar o **"makemessages"** e o **"compilemessages"**;

- Abriu os arquivos **.po** e traduziu o conteúdo dos atributos **"msgstr"**;

- Compilou os arquivos;

- Criou um formulário lá no topo do blog para o usuário escolher um idioma;

- Declarou a URL **"^i18n/"**;

- Conheceu as funções **"ugettext_lazy()"** e **"ugettext()"**, e descobriu que a primeira é **preguiçosa**, e a segunda é **proativa**... mas às vezes a preguiçosa é melhor;

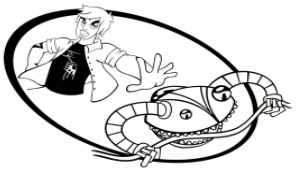
- E também agora sabe que usar um apelido **"_()"** para elas facilita a sua vida;

- E de bônus, no final, descobriu que alguns termos já são traduzidos antes que você o faça!

- Muito legal, Cartola... impressionante...

Os dois continuaram a conversar por um tempo, e Alatazan estava só um pouco preocupado com sua organização em sua vida pessoal, pois algumas contas se atrasavam e ele se esquecia disso...

Capítulo 23: Fazendo um sistema de Contas Pessoais



- Clic!
- Nada.
- Cloc! Clic!
- Nada.
- Cloc! Clic! Cloc! Clic!

Tudo escuro, nadica de nada.

- Ah, Ballzer, para com isso...

A porta se abriu um pouco mais. Continua tudo escuro.

- Paw! Hoje é dia 9! Eu não podia ter esquecido, hoje não!

Alatazan se sentou na beirada da porta com a cabeça entre as mãos, e Ballzer lhe apareceu do escuro, mastigando uns papéis...

- Vem cá seu pirralho, você ainda está mastigando o resto das minhas contas, está querendo ficar sem água também?

Ballzer não deu a mínima para a questão da água, mas fingiu preocupação...

Já não bastava o dinheiro acabando e o senhorzinho do **"Compra-se Ouro"** cada dia mais desconfiado, agora Alatazan teria também de lidar com a companhia elétrica pra resolver aquele impasse. Mas tudo foi por falta de alguma coisa pra fazê-lo se lembrar do vencimento de suas contas...

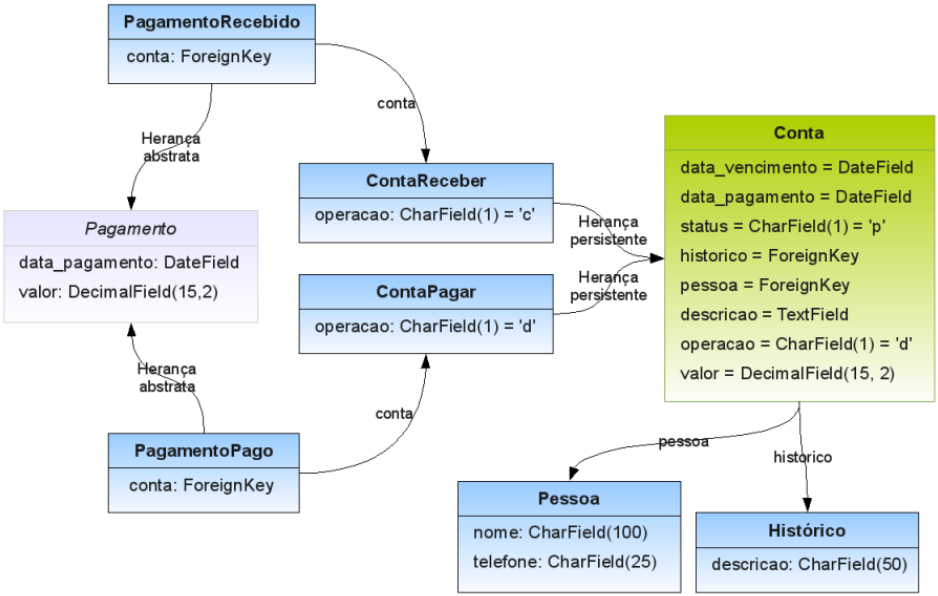
... para pagar e receber também, o Romão do parque estava atrasado em 5 dias e só agora ele se lembrava disso.

O que Alatazan precisava era de um sistema simples para controlar suas **contas pessoais**.

Mergulhando no ORM do Django

Da mesma forma como aconteceu nos capítulos do *deploy* esta será uma sequência de **4 capítulos**, pois vamos percorrer por diversos aspectos da construção de sistemas no Django e aprofundar um pouco mais o conhecimento em diversas áreas que já conhecemos.

Ao final dessa brincadeira, teremos uma nova aplicação com o seguinte modelo:



O centro da nossa aplicação são as classes **"ContaPagar"** e **"ContaReceber"**. São elas que estão no centro do **negócio** da aplicação.

Mas elas possuem exatamente os mesmos campos, e é muito útil ter **entradas e saídas** de um sistema contas de forma fácil de calcular.

Se você quiser por exemplo saber o saldo de suas **Contas a Pagar** com suas **Contas a Receber** em um mês, basta somar tudo, sendo que os **valores** das Contas a Pagar devem ser invertidos, o que pode ser facilmente resolvido com uma **multiplicação por -1**.

E é por isso que vamos usar um recurso muito bacana do Django: a **herança de modelo**.

Herança persistente

Na classe **"Conta"** serão concentrados todos os campos e a maior parte da

lógica do negócio. Esta classe deve ser **herdada** pelas classes **"ContaPagar"** e **"ContaReceber"**, que possuem somente definições diferentes para o campo da **operação financeira**, que indica se a conta é de **Débito** ou de **Crédito**.

Dessa forma as tabelas no banco de dados serão criadas assim:

```
CREATE TABLE "contas_conta" (
"id" integer NOT NULL PRIMARY KEY,
"data_vencimento" date NULL,
"data_pagamento" date NULL,
"status" varchar(1) NOT NULL,
"historico_id" integer NOT NULL REFERENCES "contas_historico" ("id"),
"pessoa_id" integer NOT NULL REFERENCES "contas_pessoa" ("id"),
"descricao" text NOT NULL,
"operacao" varchar(1) NOT NULL,
"valor" decimal NOT NULL
);


CREATE TABLE "contas_contapagar" (
"conta_ptr_id" integer NOT NULL PRIMARY KEY REFERENCES "contas_conta" ("id")

);


CREATE TABLE "contas_contareceber" (
"conta_ptr_id" integer NOT NULL PRIMARY KEY REFERENCES "contas_conta" ("id")

);
```

Esse tipo de **herança de modelo** se chama **Herança Persistente**, pois há de fato uma tabela para cada uma das **classes de modelo** envolvidas e a classe **Conta** pode ser usada normalmente para o que precisar.

Herança abstrata

Mas há outra forma de herança, chamada **Herança Abstrata**. E este é o caso das classes **"PagamentoPago"** e **"PagamentoRecebido"**.

Apesar de possuírem campos semelhantes, cada uma dessas classes é **agregada** a uma classe diferente: **"PagamentoPago"** faz parte de **Contas a Pagar**, enquanto **"PagamentoRecebido"** é parte do assunto de **Contas a Receber**.

Por isso ambas são **especializações** da classe **"Pagamento"**, que é uma classe definida como **"abstrata"**, ou seja, ela tem somente o papel de concentrar

definições comuns, mas não está presente no banco de dados e só pode ser usada como herança para outras classes.

Nesse caso, as tabelas no banco de dados ficam assim:

```
CREATE TABLE "contas_pagamentopago" (
"id" integer NOT NULL PRIMARY KEY,
"data_pagamento" date NOT NULL,
"valor" decimal NOT NULL,
"conta_id" integer NOT NULL REFERENCES "contas_contapagar" ("conta_ptr_id")
);


CREATE TABLE "contas_pagamentorecebido" (
"id" integer NOT NULL PRIMARY KEY,
"data_pagamento" date NOT NULL,
"valor" decimal NOT NULL,
"conta_id" integer NOT NULL REFERENCES "contas_contareceber" ("conta_ptr_id")
);
```

Observe que há ali somente **2 tabelas**, pois a classe **"Pagamento"** não é persistente, e seus campos estão declarados redundantemente nas tabelas das classes filhas.

Os pagamentos de uma conta são importantes porque há muitas situações onde se paga uma conta por partes... você se lembra daquela vez que estava devendo mas não tinha a grana toda, então deu a metade e na semana seguinte voltou para pagar o restante? Pois essas classes de **Pagamento** são para aquele tipo de situação.

Além do que já dissemos, há ainda as classes **"Pessoa"** e **"Historico"**, que são auxiliares no negócio, sem grandes pretensões, mas com possibilidades de se expandirem.

E por fim, há dois campos que não são esclarecidos no diagrama, que são os campos **"operacao"** e **"status"** da classe **"Conta"**. Eles são campos do tipo que muitos chamam de *flags*, pois identificam o objeto em um grupo específico. Ambos os campos serão definidos para aceitarem somente algumas opções de **escolha**, você vai ver isso daqui a pouco.

Criando a aplicação e declarando as classes

Bom, chega de blá-blá-blá e vamos ao que interessa!

Na pasta do projeto, crie a pasta da nova aplicação, chamada **"contas"**, e dentro dela um arquivo vazio, chamado **"__init__.py"**. Depois disso, abra o arquivo **"settings.py"** e acrescente a aplicação à setting **"INSTALLED_APPS"**:

```
'contas',
```

E a setting agora ficou assim:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.syndication',
    'django.contrib.flatpages',
    'django.contrib.comments',

    'blog',
    'galeria',
    'tags',
    'contas'
)
```

Salve o arquivo. Feche o arquivo.

Agora vá até a pasta da aplicação **"contas"** e crie o arquivo **"models.py"** com o seguinte código dentro:

```
# -*- coding: utf-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

class Historico(models.Model):
    class Meta:
        ordering = ('descricao',)

    descricao = models.CharField(max_length=50)

    def __unicode__(self):
        return self.descricao

class Pessoa(models.Model):
```

```
    class Meta:
        ordering = ('nome',)


    nome = models.CharField(max_length=50)
    telefone = models.CharField(max_length=25, blank=True)


    def __unicode__(self):
        return self.nome
```

Você pode notar que não há novidades aí. Criamos duas as classes de modelo que servirão às demais, com uma lógica bastante simples.

Salve o arquivo. Feche o arquivo.

Agora vamos criar um segundo arquivo na mesma pasta, chamado **"admin.py"** com o seguinte código dentro:

```
from django.contrib import admin
from django.contrib.admin.options import ModelAdmin


from models import Pessoa, Historico


class AdminPessoa(ModelAdmin):
    list_display = ('nome','telefone',)


class AdminHistorico(ModelAdmin):
    list_display = ('descricao',)


admin.site.register(Pessoa, AdminPessoa)
admin.site.register(Historico, AdminHistorico)
```

Também nenhuma novidade, apenas duas classes simples, com seus respectivos campos para a listagem.

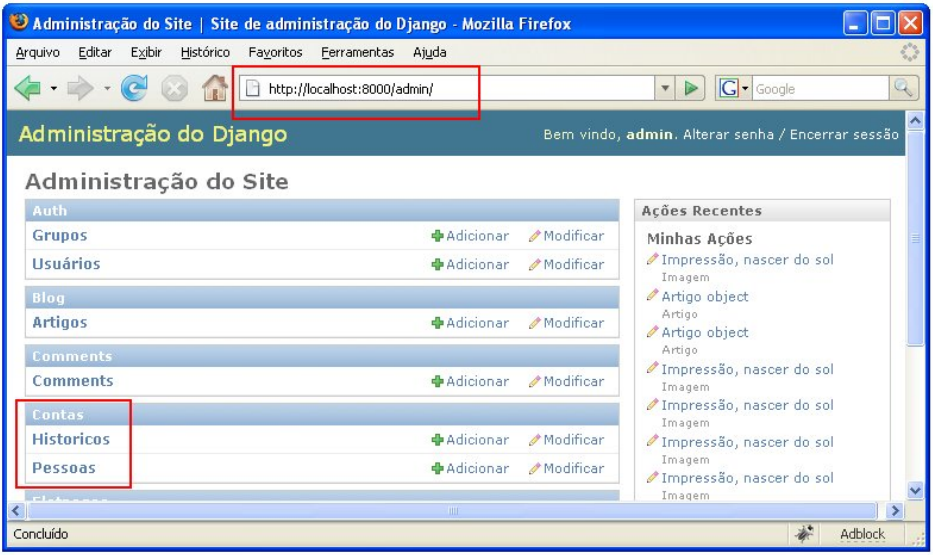
Salve o arquivo. Feche o arquivo.

Vá até a pasta do projeto e execute o arquivo **"gerar_banco_de_dados.bat"** para gerar as tabelas no banco de dados, veja o resultado:

```
Creating table contas_historico
Creating table contas_pessoa
```

Agora execute o projeto, clicando duas vezes sobre o arquivo **"executar.bat"**.

Vá ao navegador, carregue a URL do **Admin** e veja as novas classes de modelo:



Até aqui tudo bem, certo? Pisando em terreno seguro, nenhuma novidade, chega a ser monótono...

Pois agora volte à pasta da aplicação **"contas"** e abra o arquivo **"models.py"** para edição. Acrescente o seguinte código ao final do arquivo:

```
CONTA_OPERACAO_DEBITO = 'd'
CONTA_OPERACAO_CREDITO = 'c'
CONTA_OPERACAO_CHOICES = (
    (CONTA_OPERACAO_DEBITO, _('Debito')),
    (CONTA_OPERACAO_CREDITO, _('Credito')),
)

CONTA_STATUS_APAGAR = 'a'
CONTA_STATUS_PAGO = 'p'
CONTA_STATUS_CHOICES = (
    (CONTA_STATUS_APAGAR, _('A Pagar')),
    (CONTA_STATUS_PAGO, _('Pago')),
)
```

```
class Conta(models.Model):
    class Meta:
        ordering = ('-data_vencimento', 'valor')


    pessoa = models.ForeignKey('Pessoa')
    historico = models.ForeignKey('Historico')
    data_vencimento = models.DateField()
    data_pagamento = models.DateField(null=True, blank=True)
    valor = models.DecimalField(max_digits=15, decimal_places=2)
    operacao = models.CharField(
        max_length=1,
        default=CONTA_OPERACAO_DEBITO,
        choices=CONTA_OPERACAO_CHOICES,
        blank=True,
        )
    status = models.CharField(
        max_length=1,
        default=CONTA_STATUS_APAGAR,
        choices=CONTA_STATUS_CHOICES,
        blank=True,
        )
    descricao = models.TextField(blank=True)
```

E agora? Se assustou um pouco mais? Mas não se preocupe, as novidades não são muitas...

O trecho de código abaixo declara algumas variáveis úteis para identificar as opções do campo **"operacao"**. Resumindo, este campo aceita somente a escolha entre as opções **"Débito"** ou **"Crédito"**, representadas internamente pelas letras **"d"** e **"c"** respectivamente. Por exemplo: quando você escolher a opção **"Débito"** na verdade o valor salvo no banco de dados será **"d"**.

```
CONTA_OPERACAO_DEBITO = 'd'
CONTA_OPERACAO_CREDITO = 'c'
CONTA_OPERACAO_CHOICES = (
    (CONTA_OPERACAO_DEBITO, _('Debito')),
    (CONTA_OPERACAO_CREDITO, _('Credito')),
)
```

A sopa de letras em caixa alta pode assustar, mas tratam-se apenas de 2 variáveis de string com as letras **"d"** e **"c"**, e uma terceira variável com uma **tupla** contendo as 2 variáveis anteriores, seguidas por seus respectivos **rótulos**.

Agora o trecho abaixo segue a mesma sintaxe:

```
CONTA_STATUS_APAGAR = 'a'
CONTA_STATUS_PAGO = 'p'
CONTA_STATUS_CHOICES = (
    (CONTA_STATUS_APAGAR, _('A Pagar')),
    (CONTA_STATUS_PAGO, _('Pago')),
)
```

Ou seja, uma Conta pode ter dois **status** diferentes: ou a Conta está **A Pagar**, ou ela já foi **Paga**. Uma conta que não foi **completamente** paga, deve ser considerada **"A Pagar"** até que toda a dívida seja quitada.

O próximo trecho desconhecido para você será este:

```
    valor = models.DecimalField(max_digits=15, decimal_places=2)
```

O campo **"valor"** é um campo para valor **decimal** de largura definida. Isso significa que este campo pode ser preenchido com valores numéricos flutuantes com até **15 dígitos**, sendo **2 deles depois da vírgula**.

E a seguir temos os dois campos que fazem uso das variáveis que declaramos acima:

```
    operacao = models.CharField(
        max_length=1,
        default=CONTA_OPERACAO_DEBITO,
        choices=CONTA_OPERACAO_CHOICES,
        blank=True,
        )
    status = models.CharField(
        max_length=1,
        default=CONTA_STATUS_APAGAR,
        choices=CONTA_STATUS_CHOICES,
        blank=True,
        )
```

Os campos **"operacao"** e **"status"** possuem uma largura de **1 caractere** (pois os valores **"d"**, **"c"**, **"a"** e **"p"** não precisam de mais do que 1 caractere). Cada um possui seu próprio **valor default**, representado por uma daquelas variáveis. E o

argumento **"choices"** indica a **tupla** com a lista de opções de escolha.

Salve o arquivo. Feche o arquivo.

Abra novamente o arquivo **"admin.py"** pra edição e localize esta linha:

```
from models import Pessoa, Historico
```

Modifique para ficar assim:

```
from models import Pessoa, Historico, Conta
```

Agora, localize esta outra linha:

```
admin.site.register(Pessoa, AdminPessoa)
```

Acrescente **acima** dela o seguinte trecho de código:

```
class AdminConta(ModelAdmin):
    list_display = (
    'data_vencimento',
    'valor',
    'status',
    'operacao',
    'historico',
    'pessoa',
    )
    search_fields = ('descricao',)
    list_filter = (
    'data_vencimento',
    'status',
    'operacao',
    'historico',
    'pessoa',
    )
```

E por fim, acrescente esta outra linha ao final do arquivo:

```
admin.site.register(Conta, AdminConta)
```

Observe que o que fizemos é apenas o que já conhecemos: o atributo **"list_display"** define quais colunas devem ser exibidas na listagem de contas, o atributo **"search_fields"** define em quais campos as buscas devem ser feitas, e o atributo **"list_filter"** indica quais campos serão opções para filtro na listagem.

Salve o arquivo. Feche o arquivo.

Agora volte à pasta do projeto e crie as novas tabelas no banco de dados, executando o arquivo **"gerar_banco_de_dados.bat"**. O resultado será este:

```
Creating table contas_conta

Installing index for contas.Conta model
```

Vá ao navegador, no Admin, atualize a página e veja que temos a nova classe **Conta** ali.

Agora volte novamente ao arquivo **"models.py"** da pasta da aplicação **"contas"**. Acrescente as seguintes linhas de código ao final:

```
class ContaPagar(Conta):
    def save(self, *args, **kwargs):
        self.operacao = CONTA_OPERACAO_DEBITO
        super(ContaPagar, self).save(*args, **kwargs)


class ContaReceber(Conta):
    def save(self, *args, **kwargs):
        self.operacao = CONTA_OPERACAO_CREDITO
        super(ContaReceber, self).save(*args, **kwargs)
```

Veja que agora temos algo bastante diferente aqui.

Lembra-se de que falamos que as classes **"ContaPagar"** e **"ContaReceber"** são heranças persistentes da classe **"Conta"**? Pois é isso que esta linha faz:

```
class ContaPagar(Conta):
```

E esta também:

```
class ContaReceber(Conta):
```

E você deve se lembrar também que conversamos sobre cada uma dessas classes definir valores diferentes para o campo **"operacao"**, pois a classe **"ContaPagar"** deve sempre trabalhar com operação de **Débito** e a classe **"ContaReceber"** deve ser o inverso, trabalhando sempre com operação de **Crédito**.

Pois é exatamente isso que a nossa sobreposição faz aqui:

```
    def save(self, *args, **kwargs):
        self.operacao = CONTA_OPERACAO_DEBITO
        super(ContaPagar, self).save(*args, **kwargs)
```

Ou seja, ao salvar, antes de ser efetivamente gravado no banco de dados, o valor do campo **"operacao"** é forçado a ser o que queremos. O mesmo acontece aqui:

```
    def save(self, *args, **kwargs):
      self.operacao = CONTA_OPERACAO_DEBITO
        super(ContaReceber, self).save(*args, **kwargs)
```

Salve o arquivo. Feche o arquivo.

Agora abra o arquivo **"admin.py"** para edição e localize esta linha

```
from models import Pessoa, Historico, Conta
```

Modifique-a para ficar assim:

```
from models import Pessoa, Historico, Conta, ContaPagar, ContaReceber
```

E agora localize esta outra linha:

```
admin.site.register(Pessoa, AdminPessoa)
```

E acrescente este trecho de código **acima** dela:

```
class AdminContaPagar(ModelAdmin):
    list_display = (
    'data_vencimento','valor','status','historico','pessoa'
    )
    search_fields = ('descricao',)
    list_filter = (
    'data_vencimento','status','historico','pessoa',
    )
    exclude = ['operacao',]

class AdminContaReceber(ModelAdmin):
    list_display = (
    'data_vencimento','valor','status','historico','pessoa'
    )
    search_fields = ('descricao',)
    list_filter = (
    'data_vencimento','status','historico','pessoa',
    )
    exclude = ['operacao',]
```

Observe que há uma ligeira diferença entre essas duas classes e a classe **"AdminConta"** que está posicionada logo acima delas. Além de não haver o campo **"operacao"** nos atributos **"list_display"** e **"list_filter"**, há também um

novo atributo, chamado **"exclude"**, que oculta o campo **"operacao"** da página de manutenção dessas duas classes.

E agora ao final do arquivo, acrescente estas outras duas linhas:

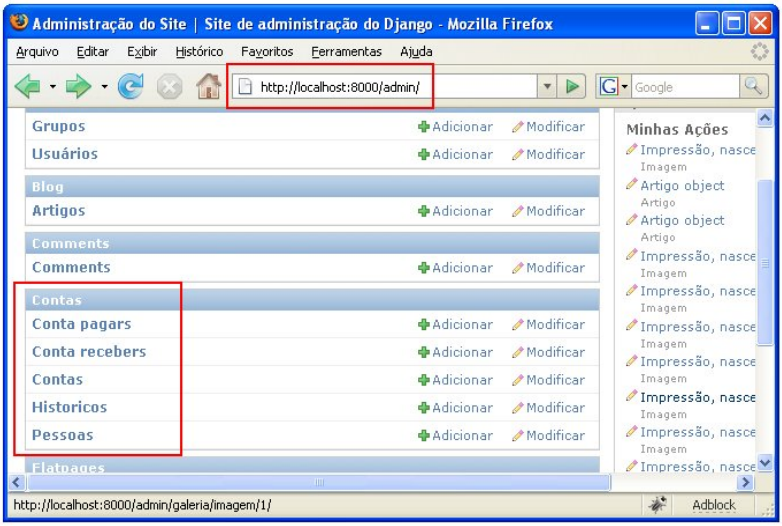
```
admin.site.register(ContaPagar, AdminContaPagar)
admin.site.register(ContaReceber, AdminContaReceber)
```

Salve o arquivo. Feche o arquivo.

Execute o arquivo **"gerar_banco_de_dados.bat"** da pasta do projeto para gerar as novas tabelas. Veja o resultado:

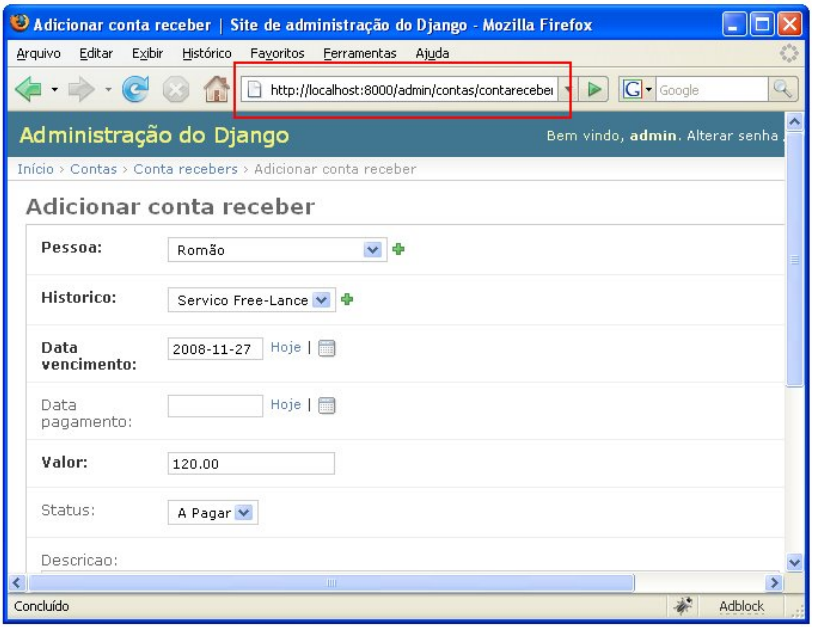
```
Creating table contas_contapagar
Creating table contas_contareceber
```

Agora volte ao navegador e carregue a **URL do Admin**:

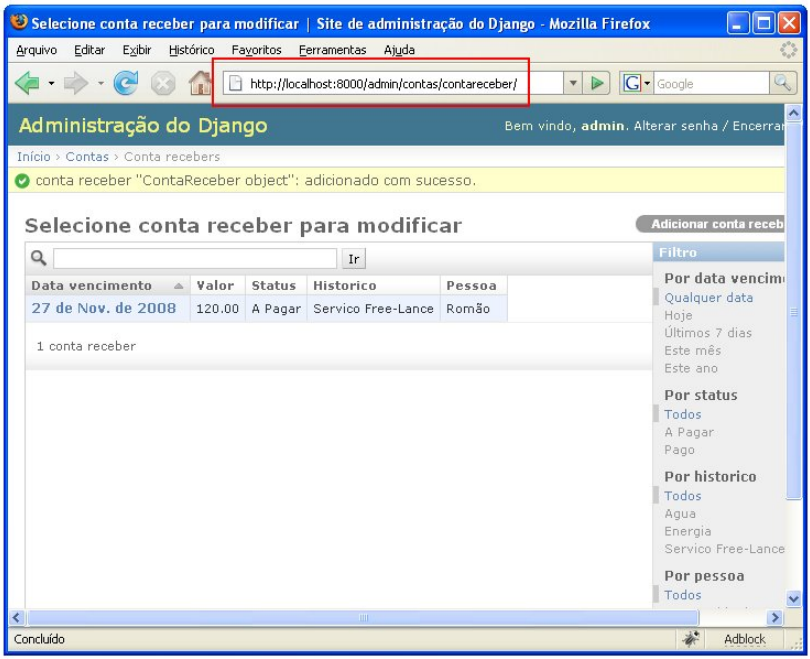


Bom, ao menos nossas classes já estão aí!

Agora crie algumas Contas a Pagar e outras a Receber, assim:

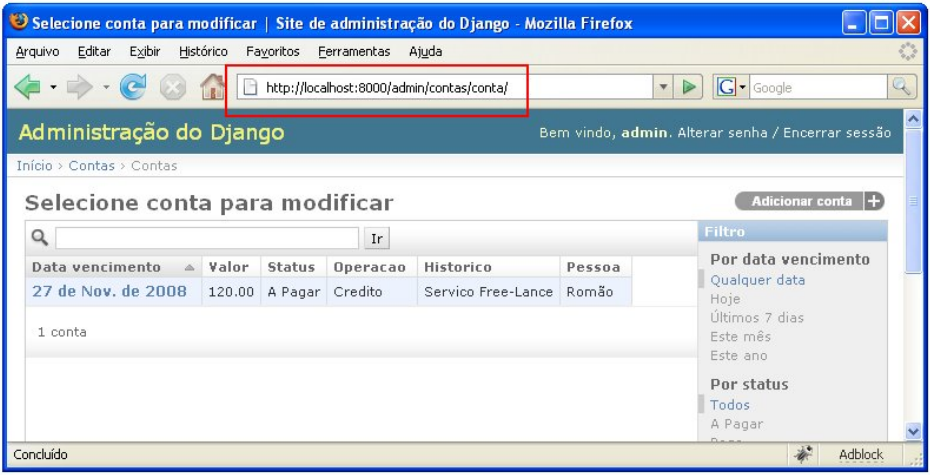


E veja como fica a página de listagem:



Como você pode ver, temos aí um conjunto de filtros bacana, e caso você vá até a página de listagem da classe **"Conta"** no Admin..

```
http://localhost:8000/admin/contas/conta/
```



... você pode ver que a mesma conta é exibida, pois de fato ela faz parte também da classe **"Conta"**!

Mas ainda faltam duas classes para criarmos: as classes de **Pagamento**.

Abra novamente o arquivo **"models.py"** da pasta da aplicação **"contas"** para edição e acrescente o seguinte trecho de código ao final do arquivo:

```
class Pagamento(models.Model):
    class Meta:
        abstract = True

    data_pagamento = models.DateField()
    valor = models.DecimalField(max_digits=15, decimal_places=2)

class PagamentoPago(Pagamento):
    conta = models.ForeignKey('ContaPagar')

class PagamentoRecebido(Pagamento):
    conta = models.ForeignKey('ContaReceber')
```

Bom, a primeira novidade que temos é este trecho de código:

```
    class Meta:
```

```
        abstract = True
```

É aquele atributo **"abstract = True"** ali que determina que a classe **"Pagamento"** é abstrata, ou seja, não possui uma tabela no banco de dados.

As demais classes **"PagamentoPago"** e **"PagamentoRecebido"** herdam essa classe e implementam cada uma seu próprio campo **"conta"**, que indica um relacionamento com sua classe de agregação.

Salve o arquivo. Feche o arquivo.

Agora abra o arquivo **"admin.py"** da mesma pasta, e vamos fazer uma coisinha diferente...

Localize esta linha:

```
from django.contrib.admin.options import ModelAdmin
```

Modifique-a para ficar assim:

```
from django.contrib.admin.options import ModelAdmin, TabularInline
```

A classe **"TabularInline"** permite a criação de recursos em **"master/detalhe"**, o que significa que você pode controlar vários itens de uma classe agregada dentro de outra. Isso é bastante divertido!

Agora localize esta outra linha:

```
from models import Pessoa, Historico, Conta, ContaPagar, ContaReceber
```

Modifique-a, para ficar assim:

```
from models import Pessoa, Historico, Conta, ContaPagar, \
    ContaReceber, PagamentoPago, PagamentoRecebido
```

Dê um pouco de atenção ao último caractere da primeira linha. É uma barra invertida (**"\"**).

No Python, tudo aquilo que não está dentro de algum delimitador específico, ao fazer um salto de linha para sua continuidade é preciso que o salto de linha seja feito com uma **barra invertida**.

Por exemplo: o seguinte caso não precisa da barra invertida, pois está entre os delimitadores de **parênteses**:

```
tupla = (
'verinha','waltinho','mychell','mychelle',
'miltinho','leni','dinha','davi'
)
```

Nem mesmo este caso abaixo, que está entre **chaves**:

```
lista = [
```

```
'verinha','waltinho','mychell','mychelle',
'miltinho','leni','dinha','davi'
]
```

E nem mesmo este, entre **colchetes**:

```
mychell = {
    'nome': 'Mychell da Cruz',
    'idade': 87,
    'cidade': 'Mara Rosa',
    'estado': 'GO'
}
```

Mas neste caso, é preciso, pois não há delimitador:

```
from models import Pessoa, Historico, Conta, ContaPagar, \
    ContaReceber, PagamentoPago, PagamentoRecebido
```

Pois bem, vamos continuar então... localize esta outra linha:

```
class AdminContaPagar(ModelAdmin):
```

E acrescente este trecho **acima** dela:

```
class InlinePagamentoPago(TabularInline):
    model = PagamentoPago
```

Esta classe permite a edição ou inclusão de vários objetos da classe **"PagamentoPago"** de uma só vez, mas para isso ser possível, ela precisa ser parte de outra. Neste caso, a "outra" trata-se da classe **"AdminContaPagar"**, portanto, localize este trecho de código:

```
class AdminContaPagar(ModelAdmin):
    list_display = (
    'data_vencimento','valor','status','historico','pessoa'
)
    search_fields = ('descricao',)
    list_filter = (
    'data_vencimento','status','historico','pessoa',
)
    exclude = ['operacao',]
```

E acrescente esta linha ao final do bloco:

```
    inlines = [InlinePagamentoPago,]
```

Precisamos fazer o mesmo para a classe **"PagamentoRecebido"**. Para isso, localize a linha abaixo:

```
class AdminContaReceber(ModelAdmin):
```

E acrescente **acima** dela:

```
class InlinePagamentoRecebido(TabularInline):
    model = PagamentoRecebido
```

E a seguir, localize este bloco de código:

```
class AdminContaReceber(ModelAdmin):
    list_display = (
    'data_vencimento','valor','status','historico','pessoa'
)
    search_fields = ('descricao',)
    list_filter = (
    'data_vencimento','status','historico','pessoa',
)
    exclude = ['operacao',]
```

E acrescente a seguinte linha abaixo ao final do bloco:

```
    inlines = [InlinePagamentoRecebido,]
```

Depois de todas essas mudanças, o nosso arquivo **"admin.py"** completo ficou assim:

```
from django.contrib import admin
from django.contrib.admin.options import ModelAdmin, TabularInline


from models import Pessoa, Historico, Conta, ContaPagar, \
   ContaReceber, PagamentoPago, PagamentoRecebido


class AdminPessoa(ModelAdmin):
    list_display = ('nome','telefone',)


class AdminHistorico(ModelAdmin):
    list_display = ('descricao',)


class AdminConta(ModelAdmin):
    list_display = (
```

```
    'data_vencimento',
    'valor',
    'status',
    'operacao',
    'historico',
    'pessoa'
    )
    search_fields = ('descricao',)
    list_filter = (
    'data_vencimento','status','operacao','historico','pessoa',
    )

class InlinePagamentoPago(TabularInline):
    model = PagamentoPago

class AdminContaPagar(ModelAdmin):
    list_display = (
    'data_vencimento','valor','status','historico','pessoa'
)
    search_fields = ('descricao',)
    list_filter = (
    'data_vencimento','status','historico','pessoa',
)
    exclude = ['operacao',]
    inlines = [InlinePagamentoPago,]

class InlinePagamentoRecebido(TabularInline):
    model = PagamentoRecebido

class AdminContaReceber(ModelAdmin):
    list_display = (
    'data_vencimento','valor','status','historico','pessoa'
    )
    search_fields = ('descricao',)
```

```
    list_filter = (
    'data_vencimento','status','historico','pessoa',
    )
    exclude = ['operacao',]
    inlines = [InlinePagamentoRecebido,]


admin.site.register(Pessoa, AdminPessoa)
admin.site.register(Historico, AdminHistorico)
admin.site.register(Conta, AdminConta)
admin.site.register(ContaPagar, AdminContaPagar)
admin.site.register(ContaReceber, AdminContaReceber)
```

Como pode ver, o arquivo esticou-se rapidinho.

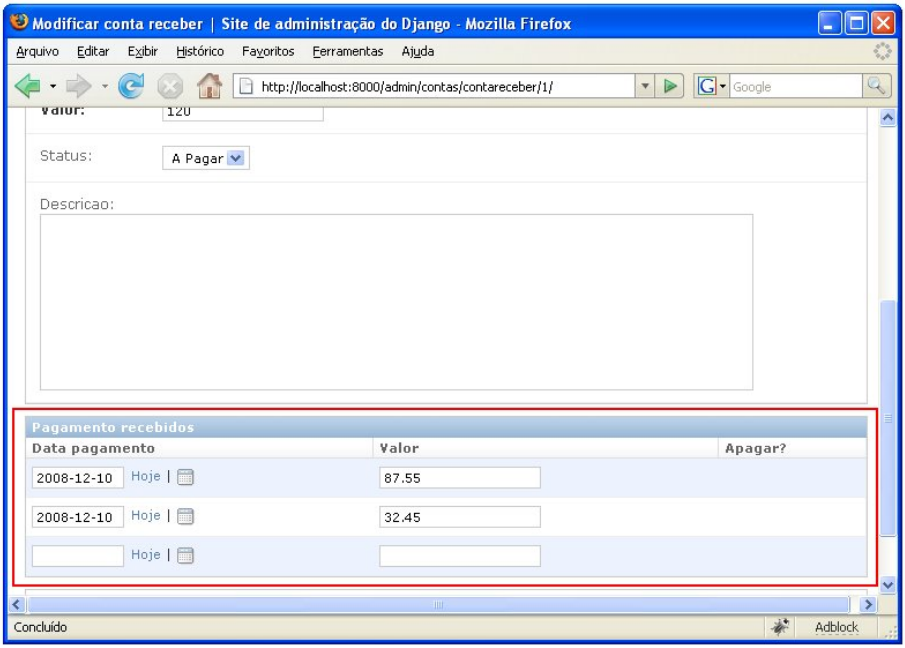
Salve o arquivo. Feche o arquivo. Execute o arquivo **"gerar_banco_de_dados.bat"** da pasta do projeto para criar as novas tabelas no banco de dados. Veja o resultado:

```
Creating table contas_pagamentopago
Creating table contas_pagamentorecebido
Installing index for contas.PagamentoPago model
Installing index for contas.PagamentoRecebido model
```

Agora com o banco de dados devidamente atualizado, vá até esta URL no Admin:

```
http://localhost:8000/admin/contas/contareceber/1/
```

Agora você pode informar a data e o valor de alguns pagamentos, como pode ver abaixo:



Incrível, não? Agora vamos fazer uma coisa a mais...

Agrupando a listagem do admin por data

Uma coisa muito bacana do Admin é que é possível organizar a listagem por data, de forma que você pode visualizar os dados pelo **ano**, **mês** e **dia** de uma forma amigável.

Para fazer isso, abra o arquivo **"admin.py"** da pasta da aplicação **"contas"** para edição e localize a seguinte linha:

```
    inlines = [InlinePagamentoPago,]
```

Observe bem que existe outra linha semelhante a ela, mas estamos falando aqui da classe **"AdminContaPagar"**. Acrescente a seguinte linha de código abaixo dela:

```
    date_hierarchy = 'data_vencimento'
```

Agora localize esta outra:

```
    inlines = [InlinePagamentoRecebido,]
```

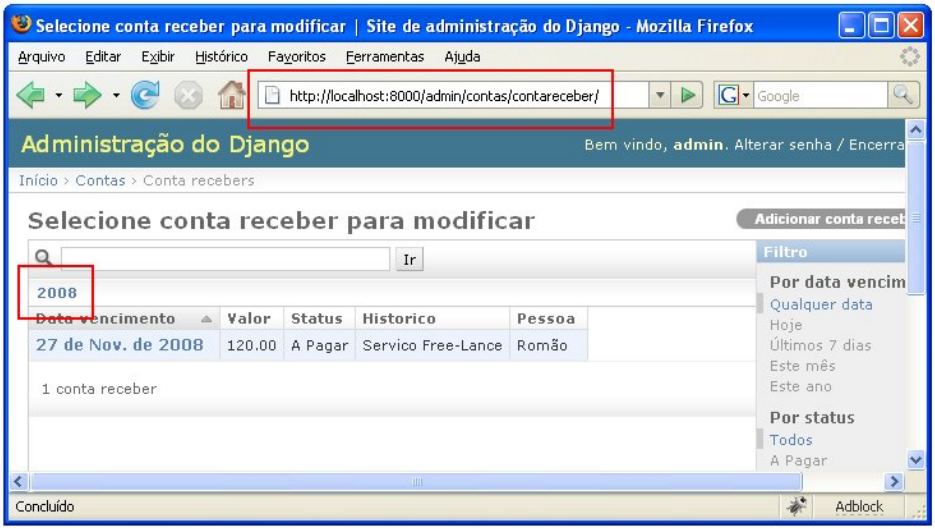
E acrescente esta abaixo dela:

```
    date_hierarchy = 'data_vencimento'
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, nesta URL do Admin:

```
http://localhost:8000/admin/contas/contareceber/
```

Observe que a página mudou levemente, veja:



Ao clicar sobre o ano, serão exibidos os meses daquele ano, e assim por diante, sempre limitados somente aos dados disponíveis.

A última linha é a que mais importa...

Isso facilitou para Alatazan com a nova tarefa que Romão - seu cliente de um pequeno sistema financeiro - pediu. O que ele queria era um sistema, sem grandes complicações, e o Admin casa perfeitamente com isso.

- Humm... então vou anotar aqui um resumo do que aprendi hoje...
  - Há dois tipos de herança de modelos: **Herança Persistente** e **Herança Abstrata**;
  - Uma classe de modelo abstrata não cria tabela no banco de dados, apenas serve de referência para suas tabelas filhas;
  - Uma classe de modelo persistente cria uma tabela e suas filhas apenas fazem um relacionamento com ela e buscam os dados de sua própria tabela. O Django se vira pra resolver isso e trazer tudo em forma de objetos redondinhos e bonitinhos;
  - Existem campos que oferecem apenas algumas opções para **escolha**. Para

esses casos, usamos o argumento **"choices"**;

- Classes filhas podem ajustar os dados das classes herdadas para se adequarem à sua realidade;

- Para ocultar um ou mais campos da página de manutenção de uma classe no Admin, usa-se o atributo **"exclude"**;

- **Barra invertida** ao final da linha indica que a linha abaixo será sua continuação, isso é legal pra evitar a escrita de linhas extensas e difíceis de compreender;

- Para fazer **master/detalhe** no Django, é preciso usar a classe **"TabularInline"** e o atributo **"inlines"** da classe de Admin;

- Usa-se o atributo **"date_hierarchy"** para organizar a listagem por um campo de data.

- É.. hoje a minha atenção ficou focada nas heranças e em algumas coisas do Admin, mas amanhã quero criar campos calculados, com somas, sumários e outras coisas que os chefes geralmente gostam...

Capítulo 24: Fazendo mais coisas na aplicação de Contas Pessoais

Alatazan estava curioso com o desempenho de *Neninha*, sua modelo predileta.

•E aí, como foi lá, Nena?

•Puxa, foi mó legal, parece que eles gostaram dela!

•É? E ela foi contratada?

• Ahhh, você sabe como é o papai né... categórico, inseguro, todo certinho... ainda vai estudar melhor...

• Mas como isso funciona, essa coisa das passarelas e tal? Tem que começar tão novinha assim?

• É sim. Olha, é como uma obra de arte, sabe... quando uma modelo abre as cortinas e sai pela passarela, ela aparece e ganha os cliques, mas o fato é que sem um **empresário** no *backstage*, desde pequena, pra acertar as coisas e dar todo o suporte, ela vai ser só mais um rosto bonitinho por trás de um balcão de uma loja de roupas...

• Sei...

• E tem a **estilista** também, sabe... toda espirituosa! Ela produz, dá vida à modelo... é ela quem faz a arte. A modelo só representa o papel.

E então? Neninha estava perto de ser a nova contratada de uma dessas agências de modelo, mas daria pra confiar naquele *Manager*? E para quem ela representaria? Bom, vamos vendo...

Buscando os números finais

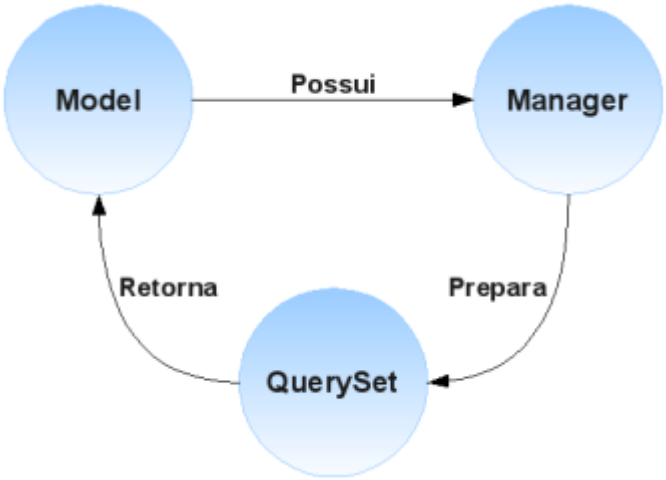
No capítulo anterior, criamos uma aplicação de **Contas Pessoais** - a pagar e a receber.

Nós falamos muito sobre heranças e algumas funcionalidades bacanas do

Admin. Mas hoje, vamos aprofundar um pouco mais ainda no ORM do Django.

Como você já sabe, o ORM é o **Mapeador Objeto/Relacional**. Trata-se de uma parte do Django que traduz as coisas de objetos Python para expressões em linguagem SQL, de forma a persistir as informações no banco de dados.

O ORM possui uma **trindade** fundalmental:



Através dos vários capítulos que se passaram, falamos em muitas ocasiões sobre classes de modelo, aquelas que herdam da classe **"django.db.models.Model"**, mas nada falamos sobre o Manager e a QuerySet.

Pois vamos falar agora!

O Manager

Toda classe de modelo possui um Manager padrão. Ele se trata do atributo **"objects"** da classe, veja aqui um exemplo onde o Manager está presente e você nem sabia disso:

```
def albuns(request):
    lista = Album.objects.all()
    return render_to_response(
        'galeria/albuns.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Observe ali esta linha:

```
    lista = Album.objects.all()
```

Sim, aquele **"objects"** ali se trata do Manager da classe **"Album"**!

E sabe o que ele faz? Ele é o ponto de apoio da classe. Veja, na grande maioria das situações em que usamos a classe de modelo diretamente, ele está lá, veja:

Criando um objeto...

```
novo_album = Album.objects.create(nome='Meu novo album')
```

Carregando um objeto:

```
album = Album.objects.get(id=1)
```

Carregando ou criando um objeto:

```
novo_album, novo = Album.objects.get_or_create(
    nome='Meu novo album'
)
```

Carregando uma lista de objetos:

```
lista = Album.objects.all()
```

Veja que lá está o Manager... nem um pouco popular e aparecido quanto o Model, mas ele está sempre lá, dando todo o suporte necessário e dando liga às coisas.

A QuerySet

E existe também a QuerySet! Ela é de fato quem dá vida ao Modelo, sempre atendendo prontamente aos pedidos do Manager.

A QuerySet efetiva as **persistências** e **consultas** no banco de dados, e é ela quem de fato dá ação à coisa e faz a arte na comunicação com o banco de dados... veja novamente os exemplos acima... ela também está lá!

O método **"create()"** é passado do Manager **"objects"** para sua QuerySet padrão:

```
novo_album = Album.objects.create(nome='Meu novo album')
```

Aqui novamente: o método **"get()"** é passado à QuerySet padrão do Manager:

```
album = Album.objects.get(id=1)
```

A mesma coisa: o método **"get_or_create()"** também é passado à QuerySet padrão do Manager:

```
novo_album, novo = Album.objects.get_or_create(
    nome='Meu novo album'
```

```
)
```

E aqui mais uma vez:

```
lista = Album.objects.all()
```

Você pode notar que na maioria das operações que tratam uma classe de modelo **não instanciada** em sua relação com o banco de dados, lá estarão presentes também a organização do **Manager** e a arte da **QuerySet**.

As QuerySets são extremamente **criativas e excitantes** porque elas possuem métodos que podem ser encadeados uns aos outros, de forma a criar uma complexa e comprida sequência de filtros e regras, que somente quando é de fato requisita, é transformada em uma expressão SQL e efetivada ao banco de dados.

Agora vamos ver essas coisas na prática?

Criando o primeiro Manager

Para começar, vá até a pasta da aplicação **"contas"** e abra o arquivo **"models.py"** para edição. Localize a seguinte linha:

```
class Historico(models.Model):
```

Acima dela, acrescente o seguinte trecho de código:

```
class HistoricoManager(models.Manager):
    def get_query_set(self):
        query_set = super(HistoricoManager, self).get_query_set()

        return query_set.extra(
            select = {
        '_valor_total': """select sum(valor) from contas_conta
        where contas_conta.historico_id = contas_historico.id""",
                }
            )
```

Opa! Mas espera aí, que tanto de novidades de uma vez só, vamos com calma né!

Ok, então vamos linha por linha...

Observe bem esta linha abaixo. Estamos criando ali um **Manager** para a classe **"Historico"**. Mas porquê criar um manager se ela já possui um?

É que o manager padrão tem um comportamento padrão, e nós queremos mudar isso. Sabe o que estamos indo fazer aqui? Nós vamos criar um novo **campo**

calculado, que irá carregar o **valor total** das **contas a pagar e a receber** ligadas ao objeto de **Histórico**!

Todo Manager deve herdar a classe **"models.Manager"**:

```
class HistoricoManager(models.Manager):
```

Este método que declaramos, **"get_query_set"** retorna a QuerySet padrão toda vez que um método do Manager é chamado para ser passado a ela. Estamos aqui novamente mudando as coisas padrão, para ampliar seus horizontes:

```
    def get_query_set(self):
```

E a primeira coisa que o método faz é chamar o mesmo método da classe herdada, mais ou menos aquilo que fizemos em um dos capítulos anteriores, lembra-se?

```
        query_set = super(HistoricoManager, self).get_query_set()
```

Logo adiante, nós retornamos a **QuerySet**, encadeando o método **"extra()"** à festa.

O método **"extra()"** tem o papel de acrescentar informações que nenhuma das outras acrescentam. Aqui podemos acrescentar campos calculados adicionais, condições especiais e outras coisinhas assim...

```
        return query_set.extra(
```

E o argumento que usamos no método **"extra()"** é o **"select"**, que acrescenta um campo calculado ao resultado da QuerySet, veja:

```
            select = {
        '_valor_total': """select sum(valor) from contas_conta
        where contas_conta.historico_id = contas_historico.id""",
                }
```

Com esse novo campo, chamado **"_valor_total"**, podemos agora exibir a soma total do campo **"valor"** de todas as **Contas** vinculadas ao Histórico.

Está um pouco confuso? Não se preocupe, vamos seguir adiante e aos poucos as coisas vão se encaixando... você não queria entender como funciona o **mundo da moda** em apenas algumas poucas palavras né?

Agora localize esta linha, ela faz parte da classe **"Historico"**:

```
descricao = models.CharField(max_length=50)
```

Acrescente abaixo dela:

```
objects = HistoricoManager()


def valor_total(self):
```

```
    return self._valor_total
```

Veja o que fizemos:

Primeiro, efetivamos o Manager que criamos - **"HistoricoManager"** - para ser o Manager padrão da classe **"Historico"**.

Vale ressaltar que uma classe de modelo pode possuir quantos Managers você quiser, basta ir criando outros atributos e atribuindo a eles as instâncias dos managers que você criou...

```
objects = HistoricoManager()
```

E aqui nós criamos uma função para retornar o valor do campo calculado **"_valor_total"**.

Sabe porquê fizemos isso? Porque há algumas funcionalidades no Django que exigem atributos declarados diretamente na classe, e como o campo **"_valor_total"** é calculado, ele "aparece" na classe somente quando esta é instanciada, resultando de uma QuerySet.

Então o que fizemos? Nós já criamos o campo com um caractere **"_"** como prefixo, e o encapsulamos por trás de um método criado manualmente...

```
def valor_total(self):
    return self._valor_total
```

Ok, vamos agora fazer mais uma coisinha, mas não aqui.

Salve o arquivo. Feche o arquivo.

Na mesma pasta, abra agora o arquivo **"admin.py"** para edição, e localize este trecho de código:

```
class AdminHistorico(ModelAdmin):
    list_display = ('descricao',)
```

Modifique a segunda linha para ficar assim:

```
class AdminHistorico(ModelAdmin):
    list_display = ('descricao','valor_total',)
```

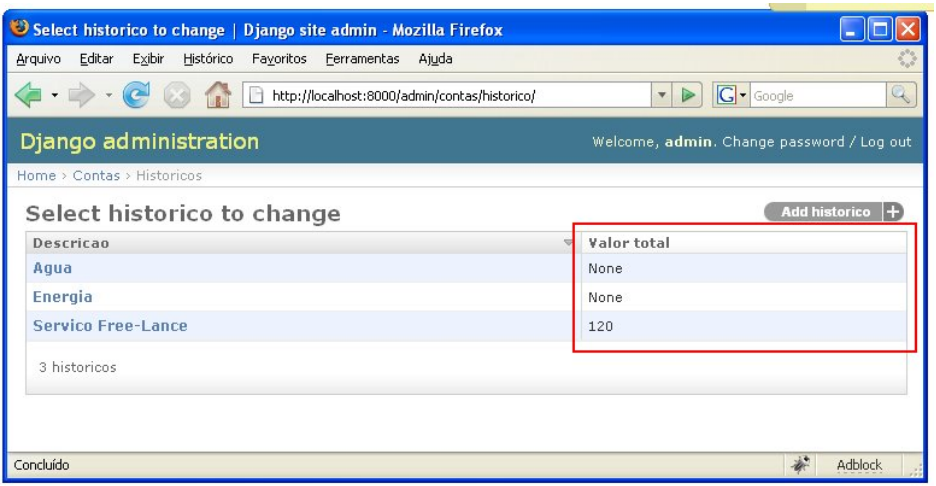
Você notou que acrescentamos o campo **"valor_total"** à listagem da classe **"Historico"**?

Salve o arquivo. Feche o arquivo.

Agora execute o projeto, clicando duas vezes sobre o arquivo **"executar.bat"** e vá até o navegador, carregando a seguinte URL:

```
http://localhost:8000/admin/contas/historico/
```

Veja como ela aparece:



Uau! Isso é legal, gostei disso! Mas vamos dar um jeito de tirar aquele **"None"** dali e deixar um **zero** quando a soma retornar um valor vazio? OK, vamos lá!

Na pasta da aplicação **"contas"**, abra o arquivo **"models.py"** para edição e localize estas linhas de código:

```
def valor_total(self):
    return self._valor_total
```

Modifique a segunda linha, para ficar assim:

```
def valor_total(self):


    return self._valor_total or 0.0
```

Isso vai fazer com que, caso o valor contido no campo calculado **"_valor_total"** seja inválido, o valor **"0.0"** retorne em seu lugar.

Salve o arquivo.

Agora vamos fazer o mesmo campo calculado para a classe **"Pessoa"**. Localize a seguinte linha:

```
class Pessoa(models.Model):
```

Acima dela, acrescente este trecho de código:

```
class PessoaManager(models.Manager):
    def get_query_set(self):
        query_set = super(PessoaManager, self).get_query_set()
```

```
        return query_set.extra(
            select = {
    '_valor_total': """select sum(valor) from contas_conta
    where contas_conta.pessoa_id = contas_pessoa.id""",
    '_quantidade_contas': """select count(valor) from contas_conta
    where contas_conta.pessoa_id = contas_pessoa.id""",
                }
            )
```

Veja que dessa vez fomos um pouquinho além, nós criamos dois campos calculados: um para **valor total** das contas e outro para sua **quantidade**. Observe que as *subselects* SQL para ambos são semelhantes, com a diferença de que a do **valor total** usa a agregação **"SUM"**, enquanto que a de **quantidade** usa **"COUNT"**.

Agora localize esta outra linha:

```
    telefone = models.CharField(max_length=25, blank=True)
```

E acrescente abaixo dela:

```
    objects = PessoaManager()


    def valor_total(self):
        return self._valor_total or 0.0


    def quantidade_contas(self):
        return self._quantidade_contas or 0
```

Você pode notar que a definimos o Manager da classe Pessoa e em seguida declaramos as duas funções que fazem acesso aos campos calculados que criamos.

Mas qual é a diferença entre retornar **"0.0"** e retornar **"0"**?

Quando um número possui uma ou mais **casas decimais**, separadas pelo ponto, o Python entende que aquele é um **número flutuante**, e já quando não possui, é um número inteiro. O **valor total** das contas sempre será um valor flutuante, mas a **quantidade** de contas sempre será em número inteiro. É isso.

Salve o arquivo. Feche o arquivo. Agora é preciso ajustar o Admin da classe **"Pessoa"**.

Abra o arquivo **"admin.py"** da mesma pasta para edição e localize o seguinte trecho de código:

```
class AdminPessoa(ModelAdmin):
```

```
    list_display = ('nome','telefone',)
```

Modifique-o para ficar assim:

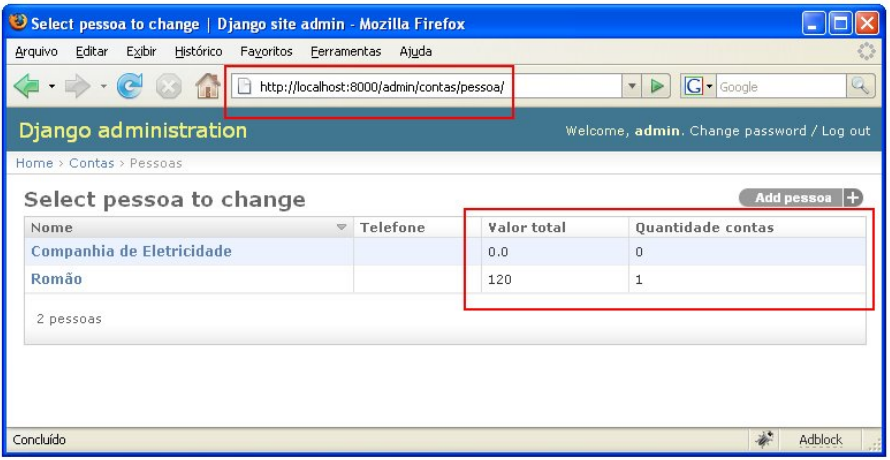
```
class AdminPessoa(ModelAdmin):
    list_display = (
    'nome','telefone','valor_total','quantidade_contas'
    )
```

Salve o arquivo. Feche o arquivo.

Volte ao navegador e carregue esta URL:

```
http://localhost:8000/admin/contas/pessoa/
```

E veja como ela se sai:

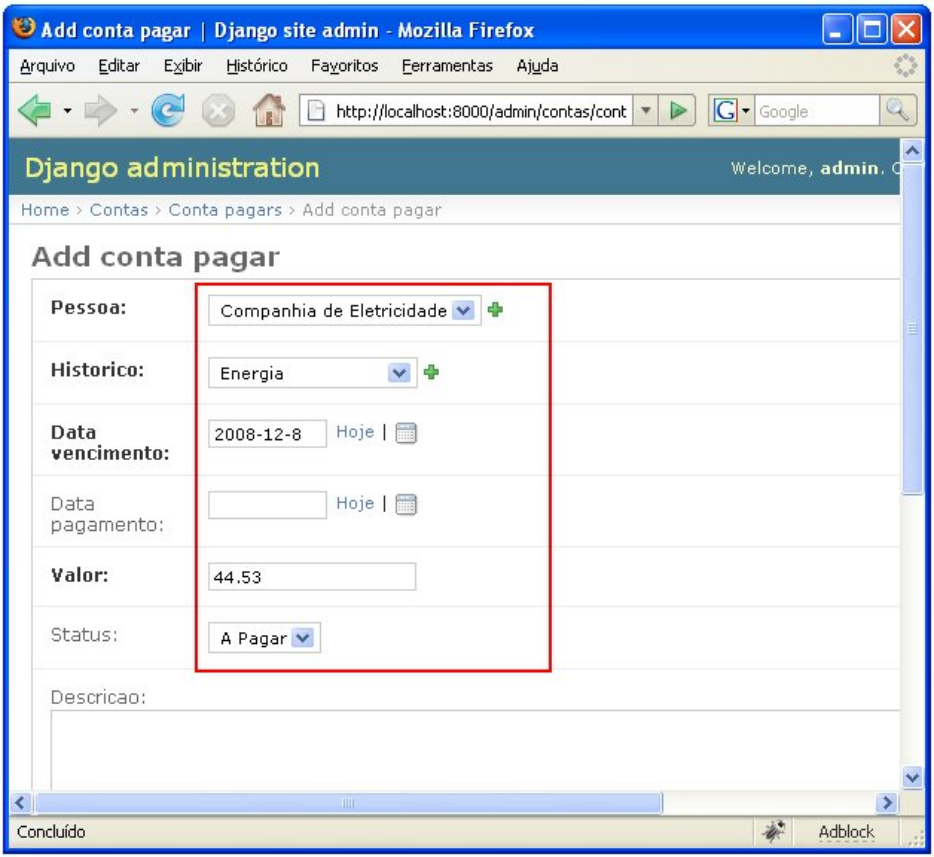


Bacana, não é? Agora vamos criar uma **conta a pagar** para ver como a coisa toda funciona, certo?

Vá a esta URL:

```
http://localhost:8000/admin/contas/contapagar/add/
```

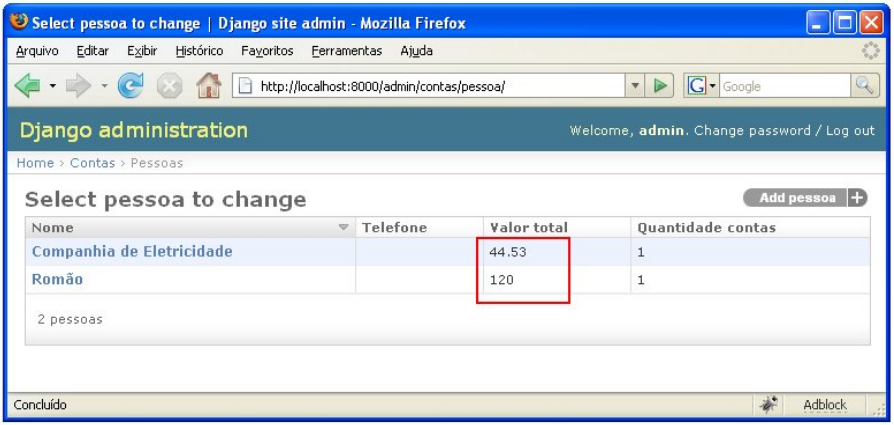
E crie uma **Conta a Pagar** como esta abaixo:



Salve a conta e volte à URL da classe **"Pessoa"** no Admin:

```
http://localhost:8000/admin/contas/pessoa/
```

E veja agora como ela está:



Isso não ficou legal... o campo de **quantidade de contas** mostra seu valor corretamente, mas veja o campo de **valor total**: se lançamos uma **Conta a Pagar**, ela deveria estar em **valor negativo**.

Então para resolver isso, abra o arquivo **"models.py"** da pasta da aplicação **"contas"** para edição e localize esta linha. Ela está dentro do método **"get_query_set()"** da classe **"HistoricoManager"**:

```
'_valor_total': """select sum(valor) from contas_conta
where contas_conta.historico_id = contas_historico.id""",
```

Modifique-a para ficar assim:

```
'_valor_total': """select sum(
valor * case operacao when 'c' then 1 else -1 end
)
from contas_conta
where contas_conta.historico_id = contas_historico.id""",
```

Observe que fazemos uma condição ali, e **multiplicamos** o campo **"valor"** por **1** ou **-1** caso a **operação financeira** seja de **crédito** ou não (de **"débito"**, no caso), respectivamente.

Agora vamos fazer o mesmo com a classe **"PessoaManager"**:

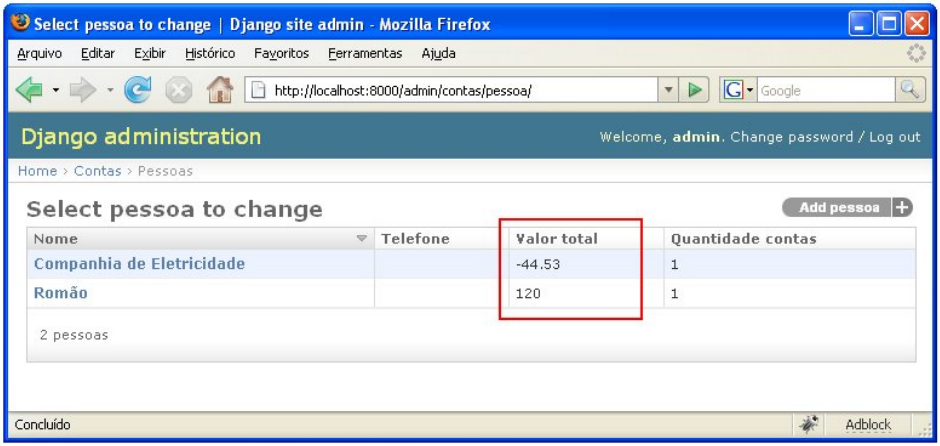
```
'_valor_total': """select sum(valor) from contas_conta
where contas_conta.pessoa_id = contas_pessoa.id""",
```

Modifique para ficar assim:

```
'_valor_total': """select sum(
valor * case operacao when 'c' then 1 else -1 end
```

```
    )
    from contas_conta
    where contas_conta.pessoa_id = contas_pessoa.id""",
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, pressione **F5** para atualizar, e veja como ficou:



Satisfeito agora?

Totalizando campos

Bom, agora que você já teve um contato razoável com um Manager e sua QuerySet padrão, que tal fazermos uma pequena mudança no Admin das **Contas**?

Vamos fazer assim: na pasta da aplicação **"contas"**, crie uma nova pasta, chamada **"templates"** e dentro dela outra pasta chamada **"admin"**.

Diferente? Pois vamos criar mais pastas: dentro da nova pasta **"admin"**, crie outra, chamada **"contas"** e dentro dela mais uma, chamada **"contapagar"**.

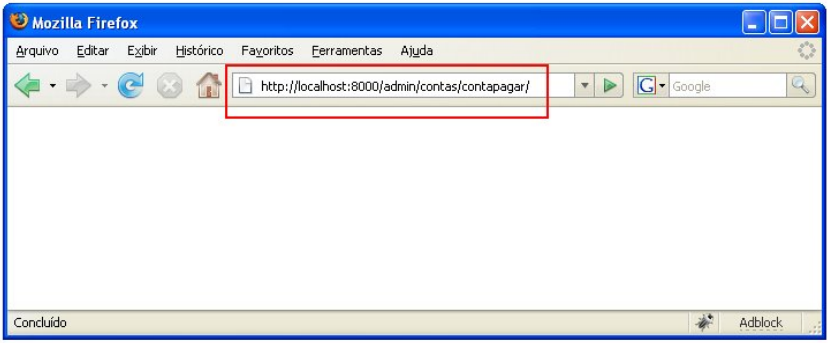
Agora dentro desta última pasta criada, crie um novo arquivo, chamado **"change_list.html"**. Deixe-o vazio por enquanto.

Agora feche a janela do Django **em execução** (aquela janela do MS-DOS que fica sempre *pentelhando* por ali) e execute novamente, para que a nossa nova pasta de **templates** faça efeito. Para isso, clique duas vezes no arquivo **"executar.bat"** da pasta do projeto.

Agora vá à URL da classe **"ContaPagar"** no Admin, assim:

```
http://localhost:8000/admin/contas/contapagar/
```

Veja:



Gostou? Lógico que não, mas o que houve de errado?

O arquivo **"change_list.html"** que criamos e deixamos vazio fez esse estrago. Sempre quando existe uma pasta de **templates** com uma pasta **"admin"**, dentro dela uma pasta com o nome da aplicação (**"contas"**) e dentro dela uma pasta com o nome da classe de modelo (**"contapagar"**), o Admin do Django tenta encontrar ali um arquivo chamado **"change_list.html"** para usar como **template da listagem** daquela classe. E o arquivo está vazio, o que mais você queria que acontecesse?

Pois então abra esse arquivo e escreva o seguinte código dentro:

```
{% extends "admin/change_list.html" %}


{% block result_list %}
<p>
  Quantidade: {{ total }}<br/>
  Soma: {{ soma|floatformat:2 }}
</p>
{% endblock result_list %}
```

Observe que no início de tudo, o template **"admin/change_list.html"** é herdado. Esse template faz parte do próprio Django e deve ser usado como base para nossas modificações. Ele é o template original que o Django sempre usa para listagens no Admin.

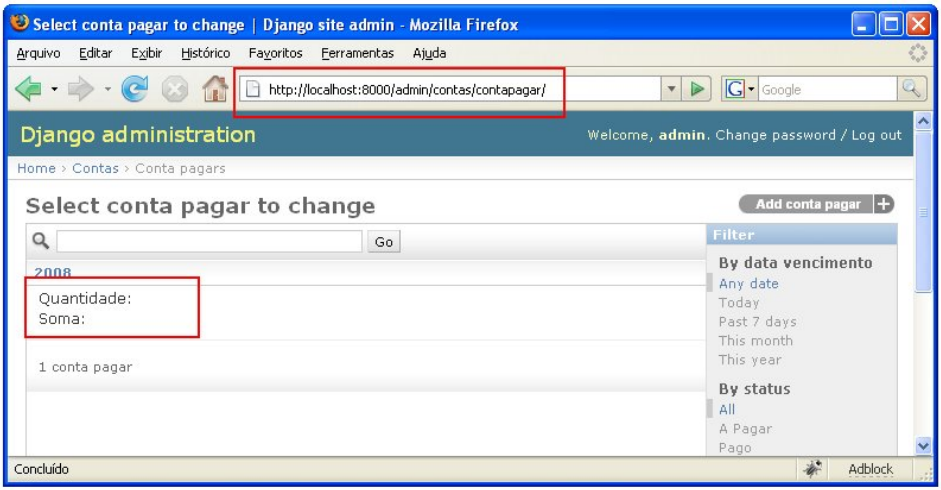
```
{% extends "admin/change_list.html" %}
```

E logo depois, nós extendemos o bloco **"result_list"** e colocamos ali duas linhas: uma para exibir a **quantidade** e outra, a **soma** das contas a pagar:

```
{% block result_list %}
```

```
<p>
  Quantidade: <b>{{ total }}</b><br/>
  Soma: <b>{{ soma|floatformat:2 }}</b>
</p>
{% endblock result_list %}
```

Salve o arquivo e volte ao navegador, atualize com **F5** e veja:



Opa, cadê as minhas contas? Não se sinta aliviado, suas contas sumiram mas continuam existindo... é que nós nos esquecemos de uma coisa importante.

Localize esta linha no template que estamos editando:

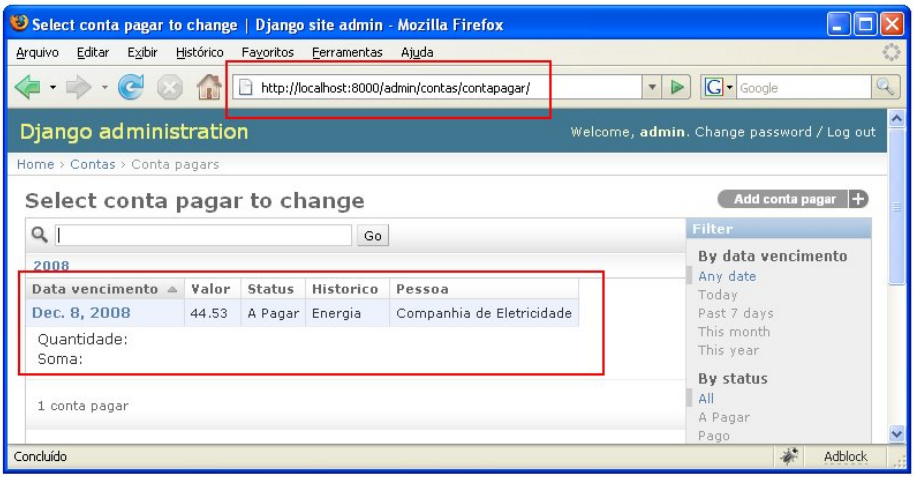
```
{% block result_list %}
```

E modifique para ficar assim:

```
{% block result_list %}{{ block.super }}
```

Isso vai evitar que perdamos o que já está lá, funcionando bonitinho.

Salve o arquivo. Feche o arquivo. Atualize o navegador com **F5** e veja:



Pronto, voltamos ao normal, com o nosso **sumário** ali, ainda que com valores vazios.

Agora vá até a pasta da aplicação **"contas"** e abra o arquivo **"admin.py"** para edição. Localize este trecho de código nele:

```
class AdminContaPagar(ModelAdmin):
    list_display = (
    'data_vencimento','valor','status','historico','pessoa'
    )
    search_fields = ('descricao',)
    list_filter = (
    'data_vencimento','status','historico','pessoa',
    )
    exclude = ['operacao',]
    inlines = [InlinePagamentoPago,]
    date_hierarchy = 'data_vencimento'
```

Aí está a classe de Admin da classe **"ContaPagar"**. Acrescente este bloco de código abaixo dela:

```
    def changelist_view(self, request, extra_context={}):
        qs = self.queryset(request)

        extra_context['soma'] = sum(
```

```
        [i['valor'] for i in qs.values('valor')]
        )
        extra_context['total'] = qs.count()

        return super(
        AdminContaPagar, self
        ).changelist_view(
        request, extra_context
        )
```

Observe que se trata de um novo método para a classe de Admin da classe **"ContaPagar"**. Este método é a *view* da URL de listagem do Admin desta classe de modelo. Na segunda linha, nós carregamos a **QuerySet** para ele, usando sua **request** (requisição):

```
    def changelist_view(self, request, extra_context={}):
        qs = self.queryset(request)
```

A QuerySet que carregamos para a variável **"qs"** já é devidamente filtrada e organizada pela chamada ao método **"self.queryset(request)"**, mas ela vai ser útil para nós.

Na declaração do método há um argumento chamado **"extra_context"** e o nosso trabalho aqui é dar a ele duas novas variáveis:

```
        extra_context['soma'] = sum(
        [i['valor'] for i in qs.values('valor')]
        )
        extra_context['total'] = qs.count()
```

Uau! Veja, o método **"values()"** de uma QuerySet retorna **somente** os valores dos campos informados como argumentos, em uma lista de dicionários, assim:

```
>>> qs.values('valor')
[{'valor': Decimal("44.53")}, {'valor': Decimal("1.0")}]
```

Nós temos ali uma *list comprehension*, criada para retornar somente o conteúdo do campo valor, assim:

```
>>> [i['valor'] for i in qs.values('valor')]
[Decimal("44.53"), Decimal("1.0")]
```

E por fim, a função **"sum"** é uma função padrão do Python que soma todos os valores de uma lista, assim:

```
>>> sum([i['valor'] for i in qs.values('valor')])
```

```
Decimal("45.53")
```

Então, a soma do campo **"valor"** será atribuída ao item **"soma"** do dicionário **"extra_context"**.

E veja que na linha seguinte nós retornamos a **quantidade** de contas a pagar:

```
        extra_context['total'] = qs.count()
```

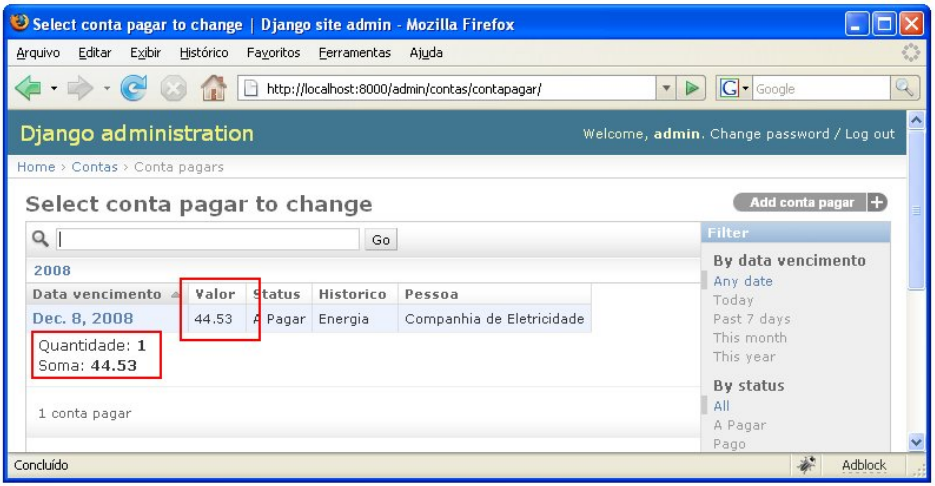
O método **"count()"** da QuerySet retorna a quantidade total dos objetos.

Por fim, a última linha é esta:

```
        return super(
        AdminContaPagar, self
        ).changelist_view(
        request, extra_context
        )
```

Como já existe um método **"changelist_view()"** na classe herdada, nós invocamos o **super()** para trazer o tratamento do método original, mas acrescentamos o dicionário **"extra_context"**, que vai levar as variáveis **"soma"** e **"quantidade"** ao contexto dos templates.

Salve o arquivo. Feche o arquivo. Volte ao navegador e pressione **F5**. Veja o resultado:



Muito bom! Agora crie outras contas a pagar, e observe como a **quantidade** e a **soma** dos valores são atualizados nessa página.

Agora, por quê não um front-end mais amigável para isso?

Alatazan mal concluiu essa sequência e já começou a ter ideias...

- Porquê não criar uma seção no site para outras pessoas organizarem suas contas pessoais?

- Bacana, camarada, como seria isso?

- Sei lá, algo assim: o cara entra, faz um cadastro e começa a organizar suas **Contas Pessoais**. Simples assim...

- Alatazan, gostei da ideia, mas vamos resumir o estudo de hoje e deixar isso para amanhã?

Nena tinha razão... havia uma coisa em comum entre eles naquele momento: **a fome roncando na barriga**...

- Ok, vamos lá:

  - Não existe Modelo sem Manager, e não existe Manager sem QuerySet. E as QuerySets trabalham principalmente com a classe de Modelo, é um ciclo que nunca se acaba;

  - A classe de modelo define como é sua **estrutura** e sua instância representa um **objeto** resultante dessa estrutura;

  - A classe de Manager **organiza** as coisas, é como o empresário de jogadores de futebol... mas o nosso Manager é justo e ético;

  - A classe de QuerySet é o **artista** da coisa, faz as coisas acontecerem de forma muito elegante;

  - Toda classe de modelo possui um Manager padrão, que possui uma QuerySet padrão;

  - O Manager padrão é atribuído ao atributo **objects** da classe de modelo, já a QuerySet padrão é retornada pelo método **"get_query_set()"** do Manager;

  - A QuerySet pode receber inúmeros métodos encadeados, como **filter()**, **exclude()**, **extra()**, **count()**, **values()**, etc. Somente quando for requisitada ela vai gerar uma expressão **SQL** como resultado disso e irá enviar ao banco de dados, da forma mais otimizada e leve possível;

  - Para interferir no funcionamento da listagem do Admin, devemos sobrepôr o método **"changelist_view()"** da classe de Admin;

  - E para mudar o template da listagem, criamos o template **"change_list.html"** dentro de uma árvore de pastas que começa da pasta de **templates**, depois uma para o próprio **admin**, outra para o **nome da**

aplicação e mais uma para o **nome da classe de modelo**.

- Algo mais?

- Ahh, camarada... eu já ia esquecendo de dizer uma coisa importate: quando uma QuerySet é requisitada ao banco de dados, o resultado do que ela resgatou de lá ficar em um **cache interno**. E caso ela seja requisitada novamente, ela será feita direto na memória, sem precisar de ir ao banco de dados de novo...

- Então vamos comer né gente!

Nos próximos capítulos vamos criar uma interface para a aplicação de **Contas Pessoais** e permitir que outros usuários trabalhem com ela.

Capítulo 25: A aplicação de Contas Pessoais sai do backstage

Alatazan esticou suas pernas para relaxar um pouco na cama. Naquele dia não haveria aula e ele sabia o que isso significava.

•Hoje eu dou uma cara para a minha aplicação web 2.0!

Nas várias vezes que acordou durante à noite ele pensou automaticamente em como faria as Contas Pessoais de seu site uma forma de ganhar dinheiro e em todas as vezes adormeceu enquanto hesitava entre anotar aquelas coisas e pensar mais um pouco.

Levantou-se, estalou as costas num movimento de fúria que faria qualquer cachorrinho tremer. Mas era só pra relaxar, nada mais...

Uma curta corrida e um pulinho, e as pontas dos pés se tocaram, quando ele percebeu que esse movimento era um tanto constrangedor no planeta onde estava vivendo.

Voltou ao chão, se refez e pegou a **conta telefônica**. Quantas páginas só pra dizer que ele devia pagar aquele valor no dia 15!

E ele se lembrou de que as contas eram mais, e algumas delas estavam um tanto desfolheadas, enfiadas como se quisessem saltar para fora de uma caixa de sapatos.

Dando uma cara à aplicação de contas pessoais

Muito do que vamos ver hoje é apenas a confirmação do que já vimos em capítulos anteriores, mas nós vamos um pouco além.

A primeira coisa a fazer é executar o seu projeto. Clique duas vezes sobre o arquivo **"executar.bat"** da pasta do projeto e pronto. A primeira coisa está feita.

Dã... mas tá, isso não pode ser considerada uma coisa a fazer, pode?

Não importa, vamos começar abrindo o arquivo **"urls.py"** para edição, para

incluir essa nova URL:

```
    (r'^contas/', include('contas.urls')),
```

Agora o arquivo inteiro ficou assim:

```
from django.conf.urls.defaults import *
from django.conf import settings

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

from blog.models import Artigo
from blog.feeds import UltimosArtigos

urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index',
        {'queryset': Artigo.objects.all(),
         'date_field': 'publicacao'}
     ),
    (r'^admin/(.*)', admin.site.root),
    (r'^rss/(?P<url>.*)/$',
     'django.contrib.syndication.views.feed',
        {'feed_dict': {'ultimos': UltimosArtigos}}
     ),
    (r'^artigo/(?P<slug>[\w_-]+)/$', 'blog.views.artigo'),
    (r'^contato/$', 'views.contato'),
    (r'^comments/', include('django.contrib.comments.urls')),
    (r'^galeria/', include('galeria.urls')),
    (r'^tags/', include('tags.urls')),
    (r'^i18n/', include('django.conf.urls.i18n')),
    (r'^contas/', include('contas.urls')),
)

if settings.LOCAL:
```

```
    urlpatterns += patterns('',
        (r'^media/(.*)$', 'django.views.static.serve',
         {'document_root': settings.MEDIA_ROOT}),
    )
```

Salve o arquivo. Feche o arquivo. Vá agora até a pasta da aplicação **"contas"** e crie o arquivo **"urls.py"**, com o seguinte código dentro:

```
from django.conf.urls.defaults import *


urlpatterns = patterns('contas.views',
    url('^$', 'contas', name='contas'),
)
```

Salve o arquivo. Feche o arquivo. Agora crie mais um novo arquivo nesta pasta, chamado **"views.py"**, com o seguinte código dentro:

```
from django.shortcuts import render_to_response
from django.template import RequestContext


from models import ContaPagar, ContaReceber, CONTA_STATUS_APAGAR


def contas(request):
    contas_a_pagar = ContaPagar.objects.filter(
        status=CONTA_STATUS_APAGAR,
        )
    contas_a_receber = ContaReceber.objects.filter(
        status=CONTA_STATUS_APAGAR,
        )


    return render_to_response(
        'contas/contas.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Bom, o que temos que novo aqui?

Nesta linha de código, nós importamos as duas classes de contas: **"ContaPagar"** e **"ContaReceber"**. E também importamos a variável que representa o status **"A Pagar"** das contas.

```
from models import ContaPagar, ContaReceber, CONTA_STATUS_APAGAR
```

Já este outro trecho de código carrega os objetos de contas a pagar e a receber, filtrados pelo campo **"status"**, que deve ter o valor contido na variável **"CONTA_STATUS_APAGAR"**. Isso significa que somente as contas que seu campo **"status"** contiver o valor informado serão retornadas para as variáveis **"contas_a_pagar"** e **"contas_a_receber"**:

```
    contas_a_pagar = ContaPagar.objects.filter(
        status=CONTA_STATUS_APAGAR,
        )
    contas_a_receber = ContaReceber.objects.filter(
        status=CONTA_STATUS_APAGAR,
        )
```

No mais, são as mesmas já conhecidas linhas de código.

Salve o arquivo. Feche o arquivo.

Agora precisamos criar o template **"contas/contas.html"** para esta view, certo? Pois ainda na pasta da aplicação **"contas"**, abra a pasta **"templates"** e crie uma nova pasta, chamada **"contas"**, e dentro dela um arquivo chamado **"contas.html"**, com o seguinte código dentro:

```
{% extends "base.html" %}


{% load i18n %}


{% block titulo %}{% trans "Contas Pessoais" %} -
{{ block.super }}{% endblock %}
{% block h1 %}{% trans "Contas Pessoais" %}{% endblock %}


{% block conteudo %}
<h2>{% trans "Contas a Pagar" %}</h2>


<ul>
  {% for conta in contas_a_pagar %}
  <li><a href="{{ conta.get_absolute_url }}">{{ conta }}</a></li>
  {% endfor %}
</ul>
```

```
<h2>{% trans "Contas a Receber" %}</h2>


<ul>
  {% for conta in contas_a_receber %}
  <li><a href="{{ conta.get_absolute_url }}">{{ conta }}</a></li>
  {% endfor %}
</ul>
{% endblock conteudo %}
```

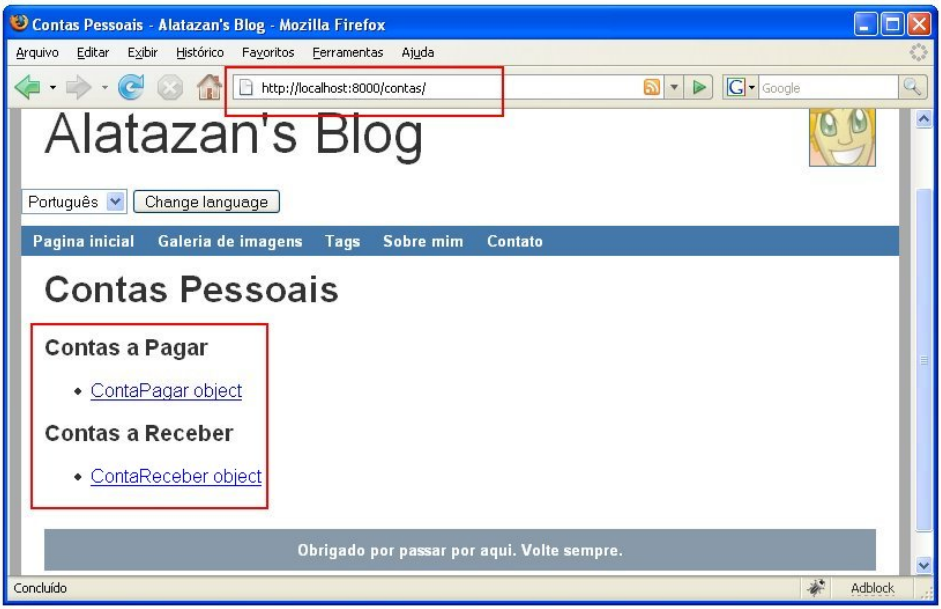
Observe bem para notar que não há novidades ali. Temos duas listagens: uma de **Contas a Pagar** e outra de **Contas a Receber**, usando as variáveis que declaramos na *view*.

Salve o arquivo. Feche o arquivo.

Agora, vá ao navegador e carregue a seguinte URL:

```
http://localhost:8000/contas/
```

Veja como ela é carregada:



Como você pode ver, fizemos progresso, mas temos que melhorar muito. Agora vá até a pasta da aplicação **"contas"** e abra o arquivo **"models.py"** para edição. Localize a seguinte linha:

```
from django.utils.translation import ugettext_lazy as _
```

Abaixo dela, acrescente esta:

```
from django.core.urlresolvers import reverse
```

Agora localize este linha:

```
    descricao = models.TextField(blank=True)
```

Acrescente esta abaixo dela:

```
    def __unicode__(self):
        data_vencto = self.data_vencimento.strftime('%d/%m/%Y')
        valor = '%0.02f'%self.valor
        return '%s - %s (%s)'%(valor, self.pessoa.nome, data_vencto)
```

Nós acrescentamos o método **"__unicode__()"** aqui porque ele será idêntico tanto para Contas a Pagar quanto para Contas a Receber, e a classe **"Conta"** concentra tudo o que há de idêntico entre as classes **"ContaPagar"** e **"ContaReceber"**.

Na linha seguinte, nós formatamos a **data de vencimento** em um formato amigável (**"dia/mês/ano"**), usando o método **"strftime()"**:

```
        data_vencto = self.data_vencimento.strftime('%d/%m/%Y')
```

E a seguir, formatamos novamente, mas dessa vez, ao invés de ser uma data, formatamos um valor monetário, garantindo duas casas decimais, veja:

```
        valor = '%0.02f'%self.valor
```

E por fim, juntamos tudo em outra formatação, para que a representação textual desse objeto fique no formato **"valor - pessoa (data de vencimento)"**:

```
        return '%s - %s (%s)'%(
        valor, self.pessoa.nome, data_vencto
        )
```

Quanta formatação hein?

Agora localize a classe **"ContaPagar"**:

```
class ContaPagar(Conta):
    def save(self, *args, **kwargs):
        self.operacao = CONTA_OPERACAO_DEBITO
        super(ContaPagar, self).save(*args, **kwargs)
```

E acrescente esse bloco de código ao final dela:

```
    def get_absolute_url(self):
        return reverse(
```

```
        'conta_a_pagar', kwargs={'conta_id': self.id}
         )
```

O que fizemos aí você já conhece: usamos a função **"reverse()"** para indicar a URL de exibição desta Conta a Pagar, que tem o nome de **"conta_a_pagar"**. E agora vamos fazer o mesmo com a Conta a Receber:

Localize a classe **"ContaReceber"**:

```
class ContaReceber(Conta):
    def save(self, *args, **kwargs):
        self.operacao = CONTA_OPERACAO_CREDITO
        super(ContaReceber, self).save(*args, **kwargs)
```

E acrescente esse bloco de código ao final dela:

```
    def get_absolute_url(self):
        return reverse(
        'conta_a_receber', kwargs={'conta_id': self.id}
        )
```

O código é muito semelhante ao da Conta a Pagar, mas desta vez indicamos a URL com o nome **"conta_a_receber"**.

Salve o arquivo. Feche o arquivo.

E como indicamos as URLs **"conta_a_pagar"** e **"conta_a_receber"**, agora precisamos que elas existam, certo? Pois então abra argora o arquivo **"urls.py"** da aplicação **"contas"** para edição e acrescente as seguintes URLs:

```
url('^pagar/(?P<conta_id>\d+)/$', 'conta', {'classe': ContaPagar},
name='conta_a_pagar'),
url('^receber/(?P<conta_id>\d+)/$', 'conta', {'classe':
ContaReceber}, name='conta_a_receber'),
```

Você pode notar que ambas as URLs fazem uso do mesmo argumento para levar o código da conta: **"conta_id"**. As duas também apontam para a mesma *view*: **"conta"**, mas daí em diante as coisas mudam, pois cada uma indica o parâmetro **"classe"** para sua respectiva classe de modelo e cada uma também possui um **nome** diferente.

Indicar parâmetros a uma URL é uma medida prática para evitar repetição de código, pois ambas as classes possuem um comportamento muito semelhante uma da outra e não há necessidade de duplicar coisas que podem ser generalizadas entre elas.

Mas falta uma coisa: importar as classes **"ContaPagar"** e **"ContaReceber"**. Para isso localize a primeira linha do arquivo:

```
from django.conf.urls.defaults import *
```

E acrescente esta abaixo dela:

```
from models import ContaPagar, ContaReceber
```

Salve o arquivo. Feche o arquivo. Agora vamos criar a *view* **"conta"**.

Para isso, ainda na pasta da aplicação **"contas"**, abra o arquivo **"views.py"** e acrescente o seguinte trecho de código ao seu final:

```
def conta(request, conta_id, classe):
    conta = get_object_or_404(classe, id=conta_id)
    return render_to_response(
        'contas/conta.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Agora para entender melhor como essa view funciona, observe as duas primeiras linhas:

```
def conta(request, conta_id, classe):
    conta = get_object_or_404(classe, id=conta_id)
```

Veja que aqui temos o argumento **"classe"**, que recebe ora a classe **"ContaPagar"**, ora **"ContaReceber"**, e quem atribui a classe a esse argumento está no código que escrevemos lá atrás, veja novamente:

```
url('^pagar/(?P<conta_id>\d+)/$', 'conta', {'classe': ContaPagar},
name='conta_a_pagar'),
url('^receber/(?P<conta_id>\d+)/$', 'conta', {'classe':
ContaReceber}, name='conta_a_receber'),
```

Observe que além do argumento **"conta_id"**, temos também o dicionário com um parâmetro dentro: **"classe"**, atribuindo a classe de modelo que esta URL indica.

Está claro? Observe com atenção, pois esta é uma prática que muitas vezes ajuda no seu dia-a-dia...

Agora localize a primeira linha do arquivo:

```
from django.shortcuts import render_to_response
```

E modifique-a, acrescentando a função **"get_object_or_404"** ao seu final, assim:

```
from django.shortcuts import render_to_response, get_object_or_404
```

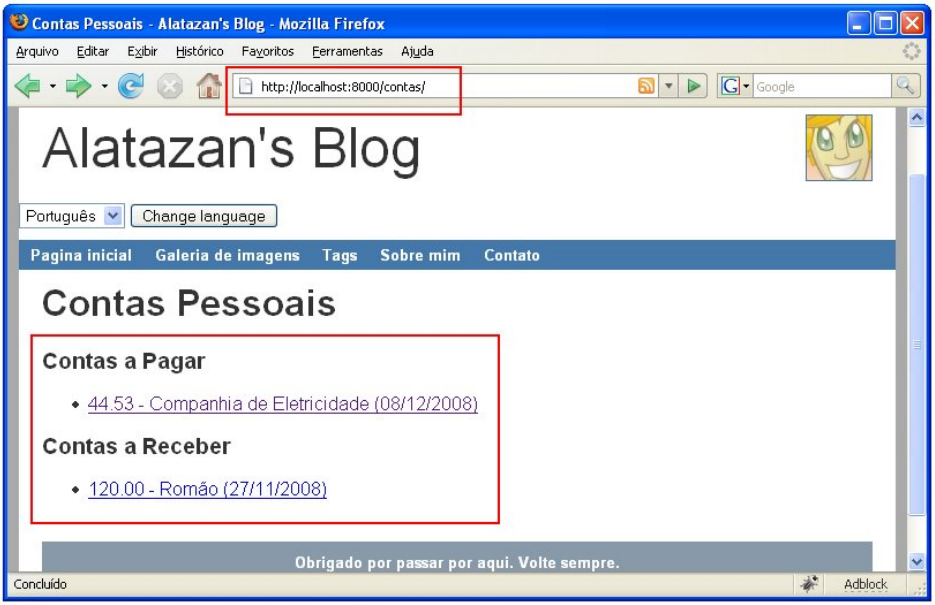
Salve o arquivo. Feche o arquivo. Já quer testar? Espere... ainda temos uma coisa a fazer.

Na pasta da aplicação **"contas"**, abra a pasta **"templates/contas"** e crie um novo arquivo, chamado **"conta.html"** com o seguinte código dentro:

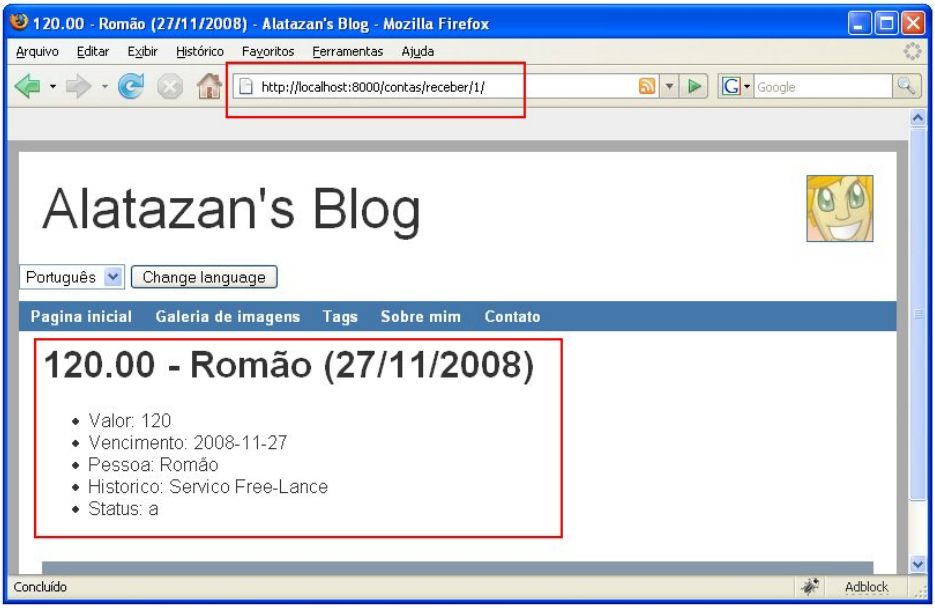
```
{% extends "base.html" %}


{% load i18n %}


{% block titulo %}{{ conta }} - {{ block.super }}{% endblock %}
{% block h1 %}{{ conta }}{% endblock %}


{% block conteudo %}
<ul>
  <li>Valor: {{ conta.valor }}</li>
  <li>Vencimento: {{ conta.data_vencimento }}</li>
  <li>Pessoa: {{ conta.pessoa }}</li>
  <li>Histórico: {{ conta.historico }}</li>
  <li>Status: {{ conta.status }}</li>
</ul>
{% endblock conteudo %}
```

Salve o arquivo (lembre-se de salvar no formato **"UTF-8"** caso esteja usando **Bloco de Notas no Windows**). Feche o arquivo.

Volte ao navegador e atualize a página com a tecla **F5**. Veja como ficou agora:



Agora clique sobre a **Conta a Receber** e veja como ela é carregada:



Estamos caminhando bem, não estamos? Vamos agora dar uma melhorada

nessa página?

Faça assim: abra para edição novamente o arquivo **"conta.html"** da pasta **"contas/templates/contas"**, partindo da pasta do projeto e localize este bloco de código:

```
<li>Valor: {{ conta.valor }}</li>
<li>Vencimento: {{ conta.data_vencimento }}</li>
<li>Pessoa: {{ conta.pessoa }}</li>
<li>Histórico: {{ conta.historico }}</li>
<li>Status: {{ conta.status }}</li>
```

Modifique para ficar assim:

```
<li>{% trans "Valor" %}: {{ conta.valor|floatformat:2 }}</li>
<li>
{% trans "Vencimento" %}:
{{ conta.data_vencimento|date:"d/m/Y" }}
</li>
<li>{% trans "Pessoa" %}:
<a href="{{ conta.pessoa.get_absolute_url }}">{{ conta.pessoa }}
</a>
</li>
<li>{% trans "Histórico" %}:
<a href="{{ conta.historico.get_absolute_url }}">
{{ conta.historico }}
</a>
</li>
<li>{% trans "Status" %}: {{ conta.get_status_display }}</li>
```

Vamos passar parte por parte:

Na primeira linha, nós formatamos o **valor** da conta para **sempre** ser exibido com **duas casas decimais**. Elas serão **arredondadas** caso sejam **mais que duas**:

```
<li>{% trans "Valor" %}: {{ conta.valor|floatformat:2 }}</li>
```

Em seguida, nós formatamos a **data de vencimento** no formato **"dia/mês/ano"**:

```
<li>
{% trans "Vencimento" %}: {{ conta.data_vencimento|date:"d/m/Y" }}
</li>
```

Nas duas linhas seguintes, nós definimos um link para os campos **"Pessoa"** e **"Histórico"**:

```
<li>{% trans "Pessoa" %}:
<a href="{{ conta.pessoa.get_absolute_url }}">{{ conta.pessoa }}
</a>
</li>
<li>{% trans "Histórico" %}:
<a href="{{ conta.historico.get_absolute_url }}">
{{ conta.historico }}
</a>
</li>
```

E em seguida... você notou que no campo **"Status"** foi mostrada uma letra **"a"**? Pois é, isso não é nada agradável... vamos mostrar seu rótulo correto! Para isso, usamos o método **"get_status_display"**, que trata-se de um método calculado, construído em memória para o campo **"status"**.

```
<li>{% trans "Status" %}: {{ conta.get_status_display }}</li>
```

Para cada um dos campos que possuem o argumento **"choices"** é oferecido um método com o nome assim: **"get_NOMEDOCAMPO_display()"**, que pode ser usado tanto em templates (removendo os parênteses) como no código em Python normalmente.

Agora vamos fazer mais. Abaixo do bloco que modificamos, acrescente mais este:

```
{% ifequal conta.status "p" %}
<li>
{% trans "Pagamento" %}: {{ conta.data_pagamento|date:"d/m/Y" }}
</li>
{% endifequal %}
```

Este código faz uma condição para ser exibido: é necessário que o **status** da conta seja **"Pago"**, representado pela condição da template tag **"{% ifequal conta.status "p" %}"**.

Agora abaixo desta linha:

```
</ul>
```

Acrescente mais estas linhas código:

```
{{ conta.descricao|linebreaks }}
```

```
{% if conta.pagamentos.count %}
<h2>{% trans "Pagamentos" %}</h2>

<table>
  <tr>
    <th>{% trans "Data" %}</th>
    <th>{% trans "Valor" %}</th>
  </tr>

  {% for pagamento in conta.pagamentos %}
  <tr>
    <td>{{ pagamento.data_pagamento|date:"d/m/Y" }}</td>
    <td>{{ pagamento.valor|floatformat:2 }}</td>
  </tr>
  {% endfor %}
</table>
{% endif %}
```

Puxa, mas quanta coisa!

Veja, nesta linha nós exibimos a **Descrição** da conta:

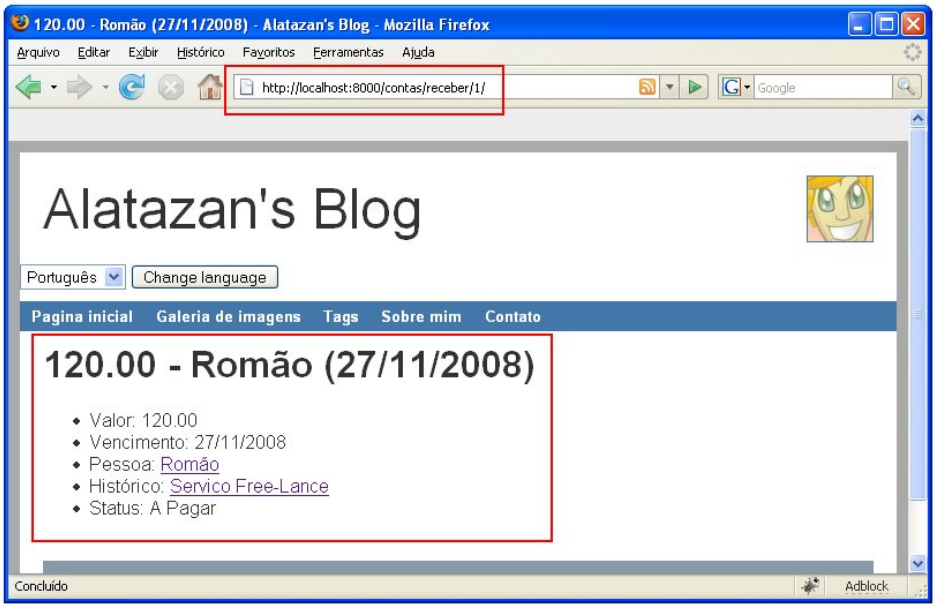
```
{{ conta.descricao|linebreaks }}
```

E aqui, iniciamos um bloco que será exibido somente se houverem sido feitos **pagamentos** na conta. A condição é: se a **quantidade** de **pagamentos** for diferente de zero (valor verdadeiro), o bloco é exibido.

```
{% if conta.pagamentos.count %}
```

E todo o restante do código que escrevemos é uma tabela com sua linha de cabeçalho e um **laço** através da template tag **{% for pagamento in conta.pagamentos %}** que vai exibir uma linha na tabela para cada um dos pagamentos.

Salve o arquivo. Feche o arquivo. Volte ao navegador e atualize a página. Veja o resultado:



Mas que bacana, tudo formatadinho... mas acontece que o atributo **"conta.pagamentos"** não existe, então, devemos criá-lo agora!

Abra o arquivo **"models.py"** da pasta da aplicação **"contas"** para edição e localize a seguinte linha. Ela faz parte da classe **"ContaPagar"**:

```
        return reverse(
        'conta_a_pagar', kwargs={'conta_id': self.id}
        )
```

Abaixo dela, acrescente o seguinte código:

```
    def pagamentos(self):
        return self.pagamentopago_set.all()
```

Isso vai fazê-la retornar a lista de pagamentos da **Conta a Pagar**.

O mesmo deve ser feito para a classe **"ContaReceber"**. Localize esta outra linha:

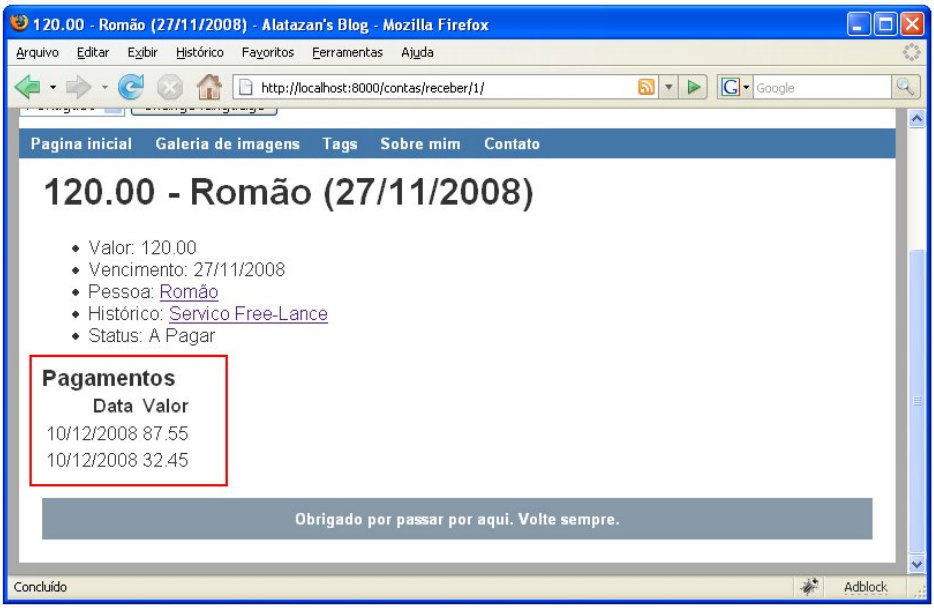
```
        return reverse(
        'conta_a_receber', kwargs={'conta_id': self.id}
        )
```

E acrescente estas abaixo dela:

```
    def pagamentos(self):
```

```
        return self.pagamentorecebido_set.all()
```

Salve o arquivo. Feche o arquivo. Agora volte ao navegador e atualize com **F5**:



Muito legal! Estamos caminhando a passos largos!

Agora precisamos de um **formulário** para fazer esses pagamentos!

Criando um formulário dinâmico para pagamentos

Então vamos lá! Volte a editar o arquivo **"conta.html"** da pasta **"contas/templates/contas"** partindo da pasta do projeto, e localize esta linha aqui:

```
{% endblock conteudo %}
```

Acima dela, acrescente estas linhas de código:

```
{% ifequal conta.status "a" %}
<hr/>
<h2>{% trans "Novo pagamento" %}</h2>

<form action="{{ conta.get_absolute_url }}pagar/" method="post">
  <table>
    {{ form_pagamento }}
  </table>
```

```
  <input type="submit" value="{% trans "Salvar pagamento" %}"/>
</form>
{% endifequal %}
```

Na primeira linha nós estabelecemos uma condição: só exibimos o formulário de pagamento se a conta ainda tiver seu **Status** como **"A Pagar"** (valor **"a"**):

```
{% ifequal conta.status "a" %}
```

Traçamos uma linha para separar visualmente o formulário do restante da página:

```
<hr/>
```

Declaramos uma tag HTML **<form>** que direciona o formulário para a URL da **conta** somada de **"pagar/"**. Observe também que ele utiliza o método **"post"**:

```
<form action="{{ conta.get_absolute_url }}pagar/" method="post">
```

E a outra coisa a observar no que fizemos é o uso do formulário dinâmico da variável **"form_pagamento"**:

```
    {{ form_pagamento }}
```

Salve o arquivo. Feche o arquivo. O próximo passo agora é fazer a variável **"form_pagamento"** existir!

Na pasta da aplicação **"contas"**, abra o arquivo **"views.py"** para edição e localize a seguinte linha de código:

```
    conta = get_object_or_404(classe, id=conta_id)
```

Acrescente esta linha abaixo dela:

```
    form_pagamento = FormPagamento()
```

Agora localize esta outra linha:

```
from models import ContaPagar, ContaReceber, CONTA_STATUS_APAGAR
```

Acrescente abaixo dela:

```
from forms import FormPagamento
```

Agora salve o arquivo. Feche o arquivo. Vamos criar a classe **"FormPagamento"**?

Na pasta da aplicação **"contas"**, crie um novo arquivo chamado **"forms.py"** com o seguinte código dentro:

```
from datetime import date
from django import forms
```

```
class FormPagamento(forms.Form):
    valor = forms.DecimalField(max_digits=15, decimal_places=2)
```

Humm... eu acho que não há novidades aí...

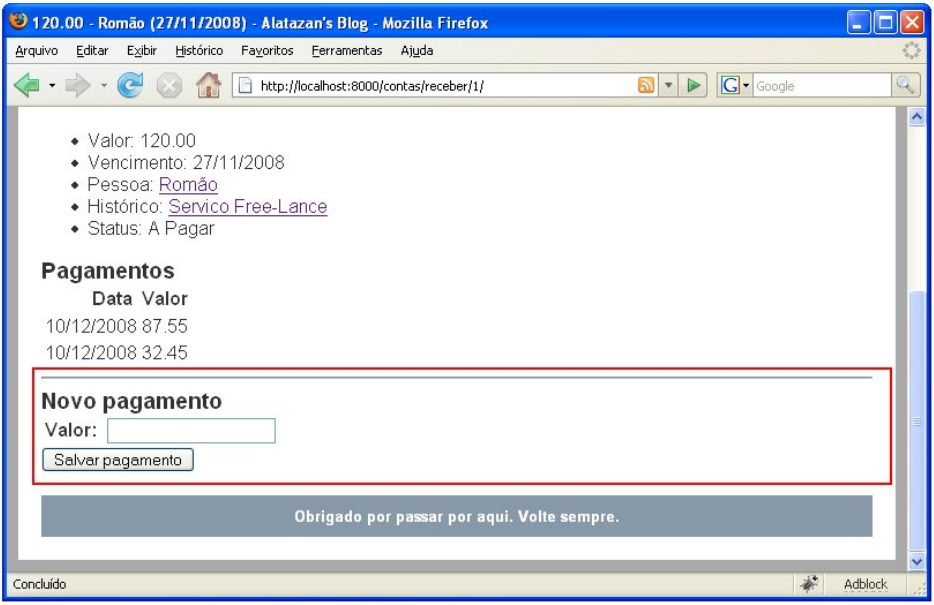
Fizemos a importação dos elementos que vamos precisar no arquivo: o objeto **"date"** para nos retornar a **data atual** e o pacote **"forms"** para criarmos o formulário dinâmico:

```
from datetime import date
from django import forms
```

Criamos a classe do formulário dinâmico e declaramos seu único campo: **"valor"**, seguindo o mesmo tipo de campo que seguimos na classe de modelo **"Pagamento"**:

```
class FormPagamento(forms.Form):
    valor = forms.DecimalField(max_digits=15, decimal_places=2)
```

Salve o arquivo. Feche o arquivo. Agora volte ao navegador e atualize a página da conta com **F5**. Veja:



Show! Nosso formulário já está aí... mas ainda sem vida hein? Pois então vamos dar vida a ele agora!

Abra o arquivo **"urls.py"** da pasta da aplicação **"contas"** e acrescente estas duas novas URLs:

```
    url('^pagar/(?P<conta_id>\d+)/pagar/$', 'conta_pagamento',
        {'classe': ContaPagar},
        name='conta_a_pagar_pagamento'),
    url('^receber/(?P<conta_id>\d+)/pagar/$', 'conta_pagamento',
        {'classe': ContaReceber},
        name='conta_a_receber_pagamento'),
```

Repare que seguimos o mesmo raciocínio das URLs da Conta a Pagar e da Conta a Receber, mas desta vez acrescentamos a palavra **"pagar/"** à expressão regular e a palavra **"_pagamento"** ao nome da *view* e ao nome da URL. Não há segredo na sopa de letrinhas!

Salve o arquivo. Feche o arquivo.

Agora vamos criar essa nova *view*! Para isso abra o arquivo **"views.py"** da mesma pasta e acrescente estas linhas de código ao final:

```
def conta_pagamento(request, conta_id, classe):
    conta = get_object_or_404(classe, id=conta_id)

    if request.method == 'POST':
        form_pagamento = FormPagamento(request.POST)

        if form_pagamento.is_valid():
            form_pagamento.salvar_pagamento(conta)

    return HttpResponseRedirect(request.META['HTTP_REFERER'])
```

Bom, veja só o que nós fizemos:

Aqui na declaração da função usamos o mesmo raciocínio da *view* **"conta()"**. Carregamos o objeto **"conta"** com base na classe informada na definição da URL:

```
def conta_pagamento(request, conta_id, classe):
    conta = get_object_or_404(classe, id=conta_id)
```

O próximo passo usou o mesmo tipo de código que usamos no formulário dinâmico de **Contato**, lembra-se do **capítulo 10**? Só aceitamos envio de dados usando método HTTP **"post"**, que aliás, usamos quando declaramos a tag HTML **<form>** neste mesmo capítulo.

```
    if request.method == 'POST':
        form_pagamento = FormPagamento(request.POST)
```

Em seguida, fazemos a **validação** do formulário dinâmico, usando o método

"is_valid()", isso não só valida os dados informados pelo usuário como também habilita o atributo **"cleaned_data"** do formulário. Depois disso o método **"salvar_pagamento()"** é chamado para o objeto **"conta"**:

```
        if form_pagamento.is_valid():
            form_pagamento.salvar_pagamento(conta)
```

E por fim, a *view* retorna um redirecionamento, usando **"HttpResponseRedirect"** e indicando a URL anterior como destino:

```
    return HttpResponseRedirect(request.META['HTTP_REFERER'])
```

O atributo **"META"** da **request** vem recheado de informações do protocolo HTTP. Isso inclui dados sobre o navegador que originou a requisição, sistema operacional, IP e outras informações. Entre elas está o item **"HTTP_REFERER"**, que carrega a URL anterior, a que o usuário estava quando clicou no botão **"Salvar pagamento"**.

E agora, para dizer ao Python quem é o tal **"HttpResponseRedirect"**, localize esta linha:

```
from django.template import RequestContext
```

E acrescente esta abaixo dela:

```
from django.http import HttpResponseRedirect
```

Salve o arquivo. Feche o arquivo. O próximo passo agora é fazer existir o método **"salvar_pagamento()"** do formulário **"FormPagamento"**.

Para isso, abra o arquivo **"forms.py"** da pasta da aplicação **"contas"** para edição e localize a classe **"FormPagamento"**:

```
class FormPagamento(forms.Form):
    valor = forms.DecimalField(max_digits=15, decimal_places=2)
```

Ao final dela, acrescente o método do qual precisamos:

```
    def salvar_pagamento(self, conta):
        return conta.lancar_pagamento(
            data_pagamento=date.today(),
            valor=self.cleaned_data['valor'],
            )
```

Veja que o método **"salvar_pagamento()"** recebe o argumento **"conta"** e chama outro método, agora do objeto **"conta"** para gravar o pagamento. O novo método se chama **"lancar_pagamento"** e recebe os argumentos **"data_pagamento"** e **"valor"** com a **data atual** e o **valor** informado pelo usuário.

Porquê fazemos isso? Porque assim podemos manter esse formulário útil tanto para Contas a Pagar quanto para Contas a Receber, mantemos também as devidas

responsabilidades separadas e nos concentramos apenas em repassar as informações das quais o formulário dinâmico é responsável, respeitando as camadas.

Salve o arquivo. Feche o arquivo.

Agora na mesma pasta abra o arquivo **"models.py"** para edição e localize este trecho de código:

```
    def pagamentos(self):
        return self.pagamentorecebido_set.all()
```

Abaixo disso, acrescente o método que acabamos de citar:

```
    def lancar_pagamento(self, data_pagamento, valor):
        return PagamentoRecebido.objects.create(
            conta=self,
            data_pagamento=data_pagamento,
            valor=valor,
            )
```

Veja que o método **"lancar_pagamento()"** recebe os argumentos **"data_pagamento"** e **"valor"** e cria o novo objeto de **"PagamentoRecebido"** com o valor dos argumentos e a referência à **conta** - que se trata do próprio objeto do método: **self**.

Faça o mesmo com a classe **"ContaPagar"**. Para isso localize o trecho de código:

```
    def pagamentos(self):
        return self.pagamentopago_set.all()
```

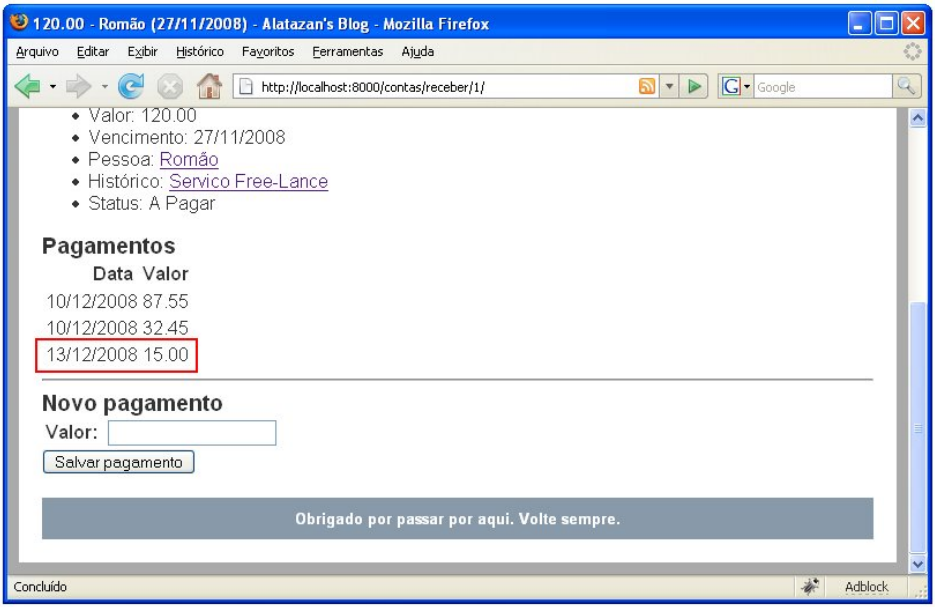
Acrescente este trecho abaixo dele:

```
    def lancar_pagamento(self, data_pagamento, valor):
        return PagamentoPago.objects.create(
            conta=self,
            data_pagamento=data_pagamento,
            valor=valor,
            )
```

A única diferença aqui é que criamos um novo objeto da classe **"PagamentoPago"** ao invés da classe **"PagamentoRecebido"**.

Salve o arquivo. Feche o arquivo.

Agora volte ao navegador, informe um valor no campo **"valor"**, clique sobre o botão **"Salvar pagamento"** e veja o que acontece:



Está lá o arquivo que você acrescentou!

Trabalhando com agrupamentos

Vamos agora criar uma página para exibir uma listagem com todas as Contas, organizadas em **páginas**.

Na pasta **"contas/templates/contas"** partindo da pasta do projeto, abra o arquivo **"contas.html"** para edição e localize esta linha:

```
<h2>{% trans "Contas a Receber" %}</h2>
```

Acima dela, acrescente esta linha:

```
<a href="{% url contas_a_pagar %}">{% trans "Ver todas" %}</a>
```

O link acima vai permitir que o usuário clique sobre ele para ver **todas as contas a pagar"**. Para fazer o mesmo com as contas a receber, localize esta outra linha de código:

```
{% endblock conteudo %}
```

E acima dela acrescente esta:

```
<a href="{% url contas_a_receber %}">{% trans "Ver todas" %}</a>
```

Salve o arquivo. Feche o arquivo. Precisamos agora criar as URLs com os nomes **"contas_a_pagar"** e **"contas_a_receber"** na aplicação **"contas"**. Para

isso, vá até a pasta da aplicação **"contas"** e abra o arquivo **"urls.py"** para edição. Acrescente as novas URLs:

```
    url('^pagar/$', 'contas_por_classe',
        {'classe': ContaPagar, 'titulo': 'Contas a Pagar'},
        name='contas_a_pagar'),
    url('^receber/$', 'contas_por_classe',
        {'classe': ContaReceber, 'titulo': 'Contas a Receber'},
        name='contas_a_receber'),
```

Veja que indicamos ambas para a *view* **"contas_por_classe"**, mas cada uma possui sua **classe** de modelo e seu nome respectivo.

A novidade está no parâmetro **"titulo"**, que leva o título da página. Logo vamos saber porquê.

Salve o arquivo. Feche o arquivo.

Agora abra o arquivo **"views.py"** da mesma pasta para edição e acrescente ao final o bloco de código abaixo:

```
def contas_por_classe(request, classe, titulo):
    contas = classe.objects.order_by('status','data_vencimento')
    titulo = _(titulo)
    return render_to_response(
        'contas/contas_por_classe.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Como você pode ver, a *view* carrega todas as contas para a variável **"contas"** usando a QuerySet de **"classe.objects.order_by('status','data_vencimento')"**, e por fim utiliza o template **"contas/contas_por_classe.html"** para retornar.

```
    contas = classe.objects.order_by('status','data_vencimento')
```

O método **"order_by('status','data_vencimento')"** define que a QuerySet deve retornar os objetos **ordenados** pelos campos **"status"** e **"data_vencimento"**, ambos em ordem ascendente.

E o argumento **"titulo"** é atribuído a uma variável com o mesmo nome, mas fazendo uso da função de **internacionalização**. Isso é porque o **título da página** deve variar dependendo da classe de modelo, veja:

```
    titulo = _(titulo)
```

Ainda precisamos importar a função de internacionalização, não é? Então

encontre esta linha:

```
from django.http import HttpResponseRedirect
```

E acrescente esta abaixo dela:

```
from django.utils.translation import ugettext as _
```

Salve o arquivo. Feche o arquivo.

Vamos agora criar o novo template! Volte à pasta **"contas/templates/contas"** e crie um arquivo chamado **"contas_por_classe.html"** com o seguinte código dentro:

```
{% extends "base.html" %}


{% load i18n %}


{% block titulo %}{{ titulo }} - {{ block.super }}{% endblock %}
{% block h1 %}{{ titulo }}{% endblock %}


{% block conteudo %}{{ block.super }}


{% for conta in contas %}
<div>
  <a href="{{ conta.get_absolute_url }}">{{ conta }}</a>
</div>
{% endfor %}


{% endblock conteudo %}
```

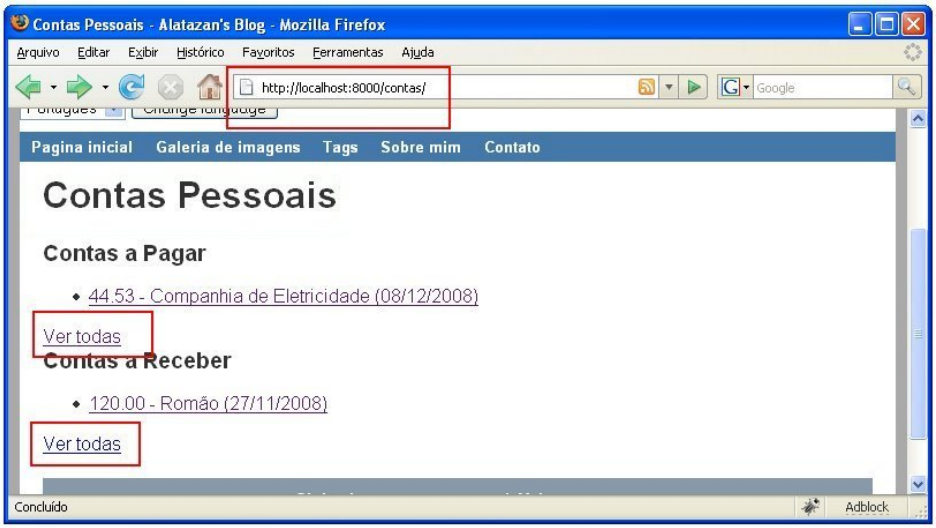
Você pode ver que usamos a variável **"titulo"** nos *blocks* **"titulo"** e **"h1"**. No mais, fazemos um laço com a template tag **{% for conta in contas %}**.

Salve o arquivo. Feche o arquivo.

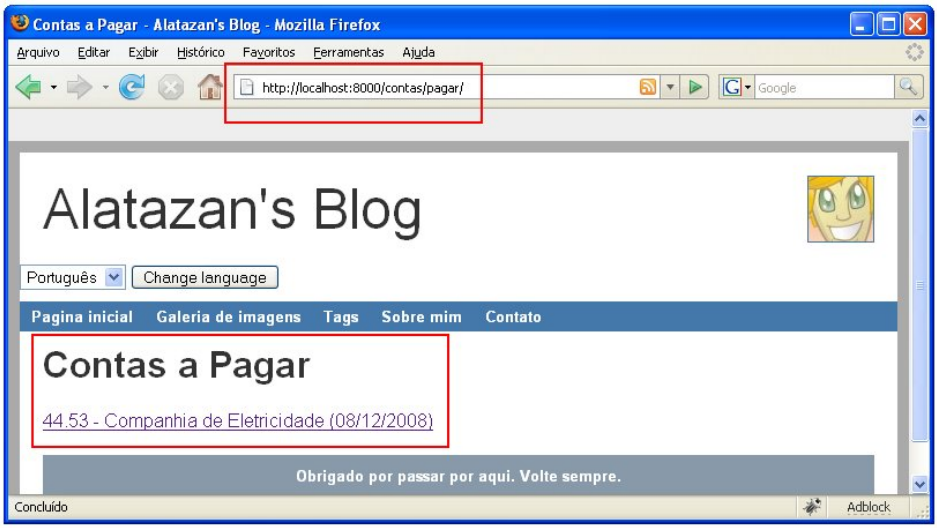
Volte ao navegador e carregue a seguinte URL:

```
http://localhost:8000/contas/
```

Veja como a página ficou agora:



Bacana! Agora clique sobre um dos links **"Ver todas"**, e veja:



Ótimo! Mas que tal agrupar as contas pelo campo de **Status**?

Então volte a editar o arquivo **"contas_por_classe.html"** da pasta **"contas/templates/contas"** localize este trecho de código:

```
{% for conta in contas %}

<div>
```

```
  <a href="{{ conta.get_absolute_url }}">{{ conta }}</a>
</div>
{% endfor %}
```

Acrescente este acima dele:

```
{% regroup contas by get_status_display as contas %}
```

Salve e volte ao navegador, atualize com **F5** e veja:

Humm... esquisito né? Porquê isso?

Veja, a template tag **{% regroup %}** agrupa uma lista de objetos por um de seus campos, atributos e métodos. Os objetos de contas possuem um método chamado **"get_status_display"** que retorna o rótulo do campo **"status"**, certo? Pois então a linha que criamos **reagrupa** a variável **"contas"** pelos valores do método **"get_status_display"** para um nome de variável: **"contas"**.

Veja que a variável que usamos como base tem o mesmo nome que a variável que damos saída: **"contas"**. Isso significa que a antiga será substituída pela nova, com o agrupamento.

A partir daí a variável **"contas"** passa a conter uma **lista de agrupamentos**. Cada grupo é um dicionário com dois itens:

- **"grouper"** - que carrega o valor agrupador - que neste caso é o rótulo do status da conta;
- **"list"** - que carrega a lista de objetos que se enquadraram dentro daquele grupo.

Agora volte ao arquivo que estamos editando e modifique este bloco de código:

```
{% for conta in contas %}
<div>
  <a href="{{ conta.get_absolute_url }}">{{ conta }}</a>
</div>
{% endfor %}
```

Para ficar assim:

```
{% for grupo in contas %}
<h2>{{ grupo.grouper }}</h2>

{% for conta in grupo.list %}
<div>
  <a href="{{ conta.get_absolute_url }}">{{ conta }}</a>
</div>
{% endfor %}

{% endfor %}
```

Veja que ali fazemos **dois laços**:

O primeiro passa a ser o laço para os grupos:

```
{% for grupo in contas %}
<h2>{{ grupo.grouper }}</h2>
```

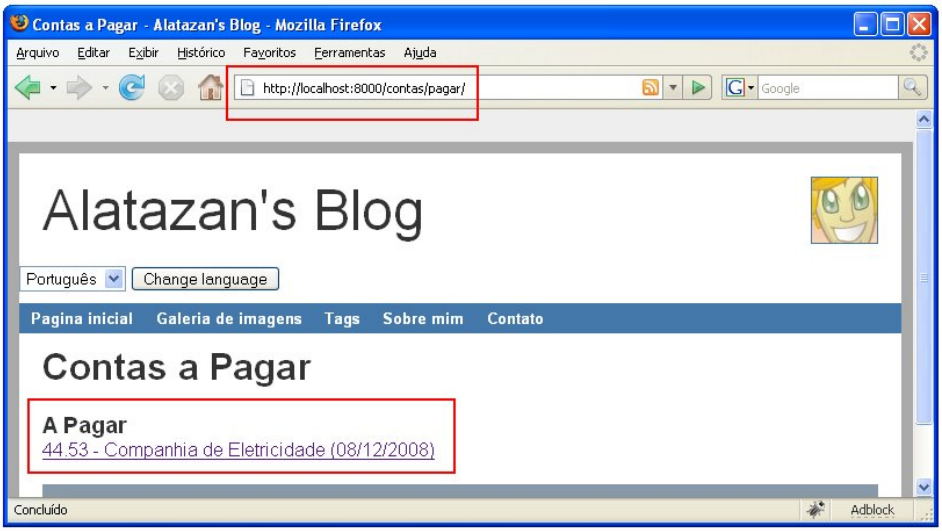
A variável **"grupo.grouper"** é exibida como **título** do grupo.

Em seguida, esta linha muda para ser um laço em **"grupo.list"**, que contém a lista de contas a partir de agora:

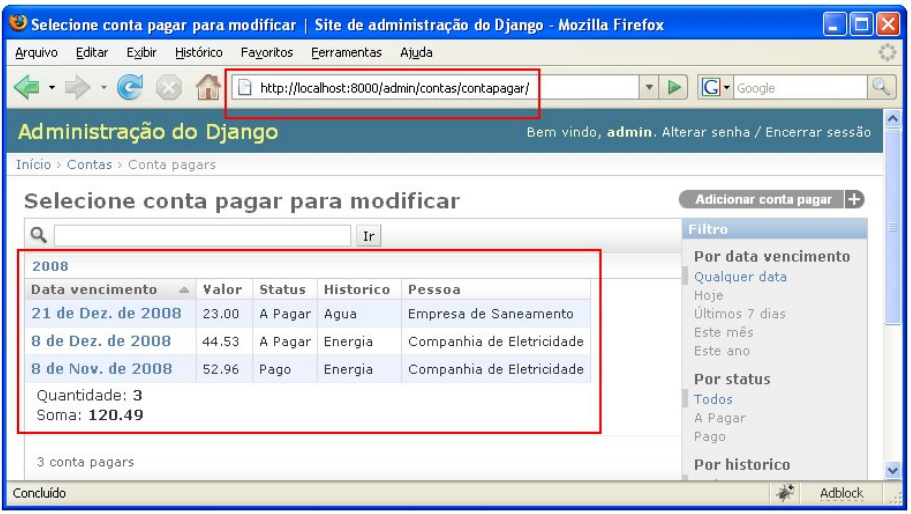
```
{% for conta in grupo.list %}
```

No mais, as coisas permanecem como estão, apenas acrescentando o fechamento dos laços.

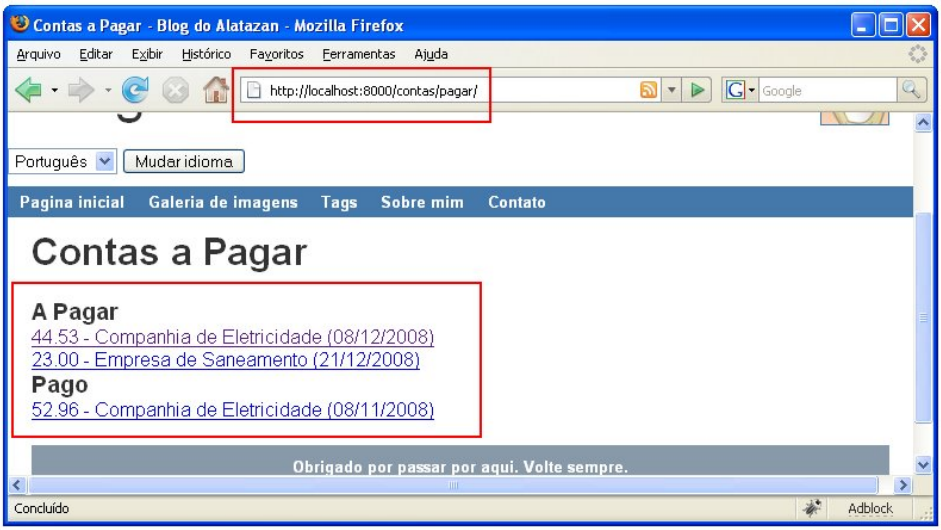
Salve o arquivo. Feche o arquivo. Atualize o navegador com **F5** e veja o resultado:



Viu? Agora que tal criar mais uma porção de contas no Admin, mudar seus **status** e voltar aqui para ver como fica? Veja um exemplo:



E a listagem equivalente na página que acabamos de construir:



É isso aí!

Trabalhando com paginação

Agora que tal organizar essas contas por **páginas de 5 em 5**? Então vamos lá!

O Django possui uma classe chamada *Paginator* que organiza listas de objetos de forma que possam ser divididas por páginas.

Na pasta da aplicação **"contas"**, abra o arquivo **"views.py"** para edição e localize a linha de código abaixo:

```
    contas = classe.objects.order_by('status','data_vencimento')
```

Acrescente estas linhas de código abaixo dela:

```
    paginacao = Paginator(contas, 5)

    pagina = paginacao.page(request.GET.get('pagina', 1))
```

O que fizemos aí foi instanciar o controlador **"paginacao"** da lista **"contas"** com páginas de 5 objetos e na segunda linha carregamos a página atual com base no número da página contido no parâmetro **"pagina"**.

Agora precisamos de importar a classe **"Paginator"**. Para isso, localize esta linha de código:

```
from django.utils.translation import ugettext as _
```

Acrescente a linha abaixo para importar a classe **"Paginator"**:

```
from django.core.paginator import Paginator
```

Salve o arquivo. Feche o arquivo.

Agora vamos ao template para habilitar a nossa paginação. Abra para edição o arquivo **"contas_por_classe.html"** da pasta **"contas/templates/contas"**, partindo da pasta do projeto. Localize esta linha:

```
{% regroup contas by get_status_display as contas %}
```

Modifique a linha para ficar assim:

```
{% regroup pagina.object_list by get_status_display as contas %}
```

Observe que trocamos o elemento **"contas"** pelo **"pagina.object_list"** e o resto permaneceu inalterado. A variável **"pagina.object_list"** traz consigo a **lista de contas** para a página atual.

Agora localize esta outra linha:

```
{% endblock conteudo %}
```

Acima dela, acrescente este trecho de código:

```
<hr/>
<div class="paginacao">
  Paginas:
  {% for pagina_numero in paginacao.page_range %}
  <a href="{{ request.path }}?pagina={{ pagina_numero }}">
  {{ pagina_numero }}
  </a>
  {% endfor %}
</div>
```

O trecho de código acima faz um laço em **"paginacao.page_range"** - a variável que carrega consigo a lista de páginas disponíveis para paginação. Dentro do laço o trabalho é exibir o número da página com seu respectivo link.

O link é composto pela URL atual em **"request.path"**, mais o parâmetro **"?pagina="**, somado ao número da página, em **"pagina_numero"**.

Mas aí há uma questão: de onde saiu a variável **"request"** que citamos em **{{ request.path }}**? É preciso fazer uma pequena modificação para que ela ganhe vida no template de forma segura!

Salve o arquivo. Feche o arquivo. Vá até a pasta do projeto e abra o arquivo **"settings.py"** para edição. Localize a seguinte linha:

```
TEMPLATE_DIRS = (
```

Acima dela, acrescente o seguinte trecho de código:

```
TEMPLATE_CONTEXT_PROCESSORS = (
```

```
    'django.core.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
    'django.core.context_processors.request',
)
```

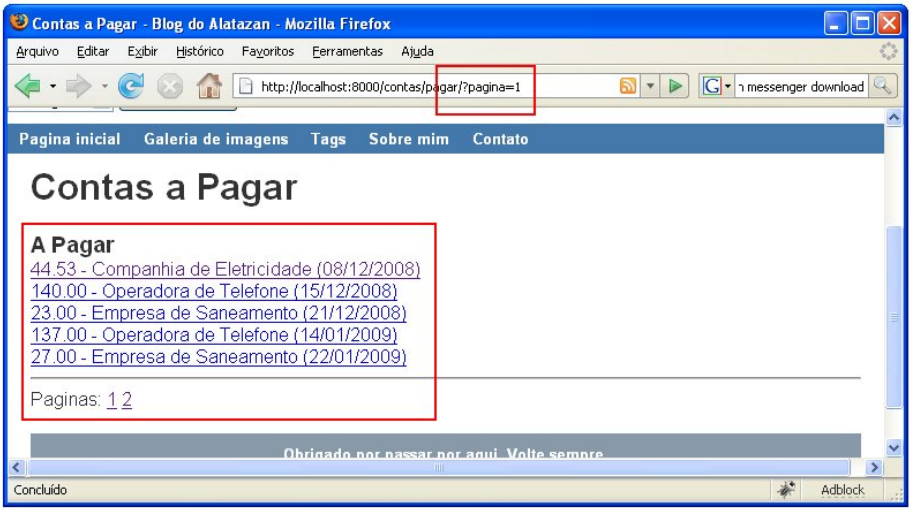
O valor padrão da setting **"TEMPLATE_CONTEXT_PROCESSORS"** é o mesmo que definimos, exceto pelo último item. Então o que fizemos na verdade foi acrescentar este item:

```
    'django.core.context_processors.request',
```

É ele quem vai fazer a variável **"request"** ser visível em qualquer *view*.

Template Context Processors são funções que retornam dicionários de variáveis para o contexto padrão dos templates, o que significa que são variáveis que estarão presentes em todos os templates mesmo que não sejam declarados nas *views*.

Salve o arquivo. Feche o arquivo. Agora vá ao navegador, atualize com **F5** e veja o resultado:



E aí, fantástico, não?

Melhorando um pouco mais e separando as contas para usuários

Depois desse ritmo alucinante, Alatazan estava cansado...

... em parte porque todas as vezes que ele imaginava um **"ahh, agora te peguei!"** para o Django, o Django respondia com um: **"hummm, vê se pegou direito o que mostrei pra você hoje!"**.

Mas em outra parte também porque sua aplicação caminhava firme para ser organizada por usuários e publicada em breve e ele não aguentava mais de ansiedade.

Mas o que Alatazan precisava rapidamente fazer era organizar as ideias sobre o aprendizado de hoje, e elas foram:

- Algumas funcionalidades da aplicação podem ser compartilhadas entre outras em comum, de forma a evitar a repetição de código;
- URLs podem passar parâmetros para *views* para que elas sejam extensíveis;
- Todos os campos que possuem a propriedade **"choices"** podem ser usados pelo método **"get***NOMEDOCAMPO***display()"** para retornar seus rótulos;
- Formulários dinâmicos podem ser usados para fazer lançamentos em classes de modelo de infinitas formas diferentes;
- A template tag **"{% regroup %}"** organiza uma lista em grupos;
- Com duas linhas cria-se a paginação para uma lista, usando a classe **"Paginator"**;
- Com mais algumas linhas se resolve a paginação no template;
- **Template Context Processors** adicionam variáveis ao contexto das templates e isso pode ser muito útil;

Alatazan parou por aqui, querendo cama e um bom chá com pão e manteiga.

O próximo passo agora é a **edição, criação e exclusão** de Contas, Pessoas e Históricos. Por fim, também organizar isso tudo para separar para usuários diferentes.

Capítulo 26: Separando as contas pessoas para usuários

- Dezessete! - a moça gritou lá na frente.

Alatazan olhou no seu papel..

"Quarenta e cinco. Hummm..."

Pensou ele...

O relógio dizia que estava ali havia uma hora e trinta e sete minutos...

Um papel na parede dizia:

"Defenda o seu! A Lei protege você! 20 minutos é o tempo máximo para espera."

Ele pensou:

- Porquê diabos eu sou obrigado a ver aquele...
- Dezoito! - gritou ela novamente.
- ... papel ali, se todo...
- Sinto muito senhora, mas a sua senha é na seção da esquerda.
- ... mês é a mesma coisa?
- Não, não atendemos senhoras debilitadas aqui. E na próxima fila o tempo começa a contar novamente, em breve você será atendida.
- Também não há banheiro público aqui.

Cada um na sua e olhe lá

Nem tudo no sistema é coletivo. Não é assim, um oba-oba... uns são seus, outros são meus... e outros, aqueles outros, aqueles são os nossos, sabe? Contas são coisas que todo mundo quer doar mas ninguém quer assumir as de outros... é assim e pronto!

Então é por isso mesmo - exatamente isso mesmo - que nós precisamos mudar essas classes de modelo das Contas Pessoais para serem estritamente separadas por usuários.

Campo para o usuário

Então, vamos começar pela classe **"Historico"**! Abra o arquivo **"models.py"** da aplicação **"contas"** para edição, e localize este bloco de código:

```
class Historico(models.Model):
    class Meta:
        ordering = ('descricao',)


    descricao = models.CharField(max_length=50)
```

Logo abaixo dele, acrescente a seguinte linha:

```
    usuario = models.ForeignKey(User)
```

Agora localize este outro bloco:

```
class Pessoa(models.Model):
    class Meta:
        ordering = ('nome',)


    nome = models.CharField(max_length=50)
    telefone = models.CharField(max_length=25, blank=True)
```

E acrescente esta abaixo dele:

```
    usuario = models.ForeignKey(User)
```

E por fim, faça o mesmo com a classe **"Conta"**:

```
class Conta(models.Model):
    class Meta:
        ordering = ('-data_vencimento', 'valor')


    pessoa = models.ForeignKey('Pessoa')
```

Logo acima do campo **"pessoa"**, acrescente a linha abaixo:

```
    usuario = models.ForeignKey(User)
```

Agora encontre esta linha entre as primeiras do arquivo:

```
from django.core.urlresolvers import reverse
```

E acrescente esta abaixo dela:

```
from django.contrib.auth.models import User
```

Salve o arquivo. Feche o arquivo. E agora, vamos ter que criar todos esses campos manualmente?

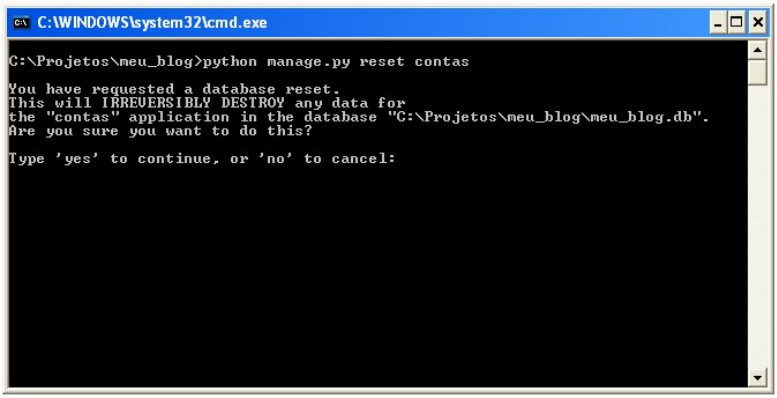
Teríamos. Mas neste caso, o campo é importante demais para passar desapercebido. Para este caso, vamos usar um outro recurso do Django que elimina as **tabelas da aplicação** e as cria novamente com os campos novos. Mas cuidado com isso, pois seus dados nesta aplicação serão perdidos, ok?

Usando o manage.py reset

Na pasta do projeto, crie um arquivo chamado **"reset_contas.bat"** com o seguinte código dentro:

```
python manage.py reset contas

pause
```

Salve o arquivo. Feche o arquivo. Agora clique duas vezes sobre o arquivo para executá-lo, e veja a mensagem que é exposta:



Em outras palavras: todos os dados da aplicação **"contas"** serão perdidos. Para confirmar, digite **"yes"** e pressione a tecla **"ENTER"**.

Os dados da aplicação **"contas"** serão perdidos mas as demais serão mantidas intactas. Por isso, vá ao **Admin** e acrescente novas **Pessoas** e **Históricos** para continuar com nosso aprendizado.

Mas antes você deve executar o projeto clicando duas vezes sobre o arquivo

"executar.bat" da pasta do projeto.

Agora nosso próximo passo será...

Criar formulários para criar e editar Contas

Agora vamos editar o arquivo **"contas.html"** da pasta **"contas/templates/contas"** e localizar esta linha:

```
<a href="{% url contas_a_pagar %}">{% trans "Ver todas" %}</a>
```

Abaixo dela, acrescente esta:

```
<a href="{% url nova_conta_a_pagar %}">{% trans "Criar nova" %}
</a>
```

Localize também esta:

```
<a href="{% url contas_a_receber %}">{% trans "Ver todas" %}</a>
```

E acrescente esta abaixo dela:

```
<a href="{% url nova_conta_a_receber %}">{% trans "Criar nova" %}
</a>
```

Com isso, agora temos *links* para criar novas contas para cada tipo: **"a Receber"** e **"a Pagar"**.

Salve o arquivo. Feche o arquivo. Pois agora vamos criar as URLs que acabamos de referenciar nos *links*.

Na pasta da aplicação **"contas"**, abra o arquivo **"urls.py"** e acrescente as seguintes URLs:

```
    url('^pagar/nova/$', 'editar_conta',
        {'classe_form': FormContaPagar,
         'titulo': 'Conta a Pagar'},
          name='nova_conta_a_pagar'),
    url('^receber/nova/$', 'editar_conta',
        {'classe_form': FormContaReceber,
         'titulo': 'Conta a Receber'},
          name='nova_conta_a_receber'),
```

Veja que continuamos apostando na estratégia de duas URLs para contas distintas, porém usando a mesma *view*. Mas desta vez usamos um parâmetro chamado **"classe_form"** ao invés de **"classe"**. Ele vai nos indicar a classe de formulário dinâmico que esta URL vai usar para validar e salvar os dados no banco

de dados.

Mas ainda não acabamos. Localize esta linha:

```
from models import ContaPagar, ContaReceber
```

E acrescente esta logo abaixo dela:

```
from forms import FormContaPagar, FormContaReceber
```

Salve o arquivo. Feche o arquivo.

Usando ModelForms

Agora, para dar vida às duas novas classes que importamos, abra o arquivo **"forms.py"** da mesma pasta e adicione estas linhas ao seu final:

```
class FormContaPagar(forms.ModelForm):
    class Meta:
        model = ContaPagar


class FormContaReceber(forms.ModelForm):
    class Meta:
        model = ContaReceber
```

Esses são os nossos dois novos formulários. Ao vincular as classes de modelo usando o atributo **"model"**, nós fizemos com que essas classes tenham todo comportamento necessário para criar e editar objetos de suas respectivas classes.

Agora encontre esta outra linha no mesmo arquivo:

```
from django import forms
```

E acrescente esta logo abaixo dela:

```
from models import ContaPagar, ContaReceber
```

Salve o arquivo. Feche o arquivo.

Ainda na mesma pasta, abra o arquivo **"views.py"** para edição e acrescente estas novas linhas de código ao seu final:

```
def editar_conta(request, classe_form, titulo, conta_id=None):
    form = classe_form()


    return render_to_response(
        'contas/editar_conta.html',
```

```
        locals(),
        context_instance=RequestContext(request),
        )
```

Salve o arquivo. Feche o arquivo.

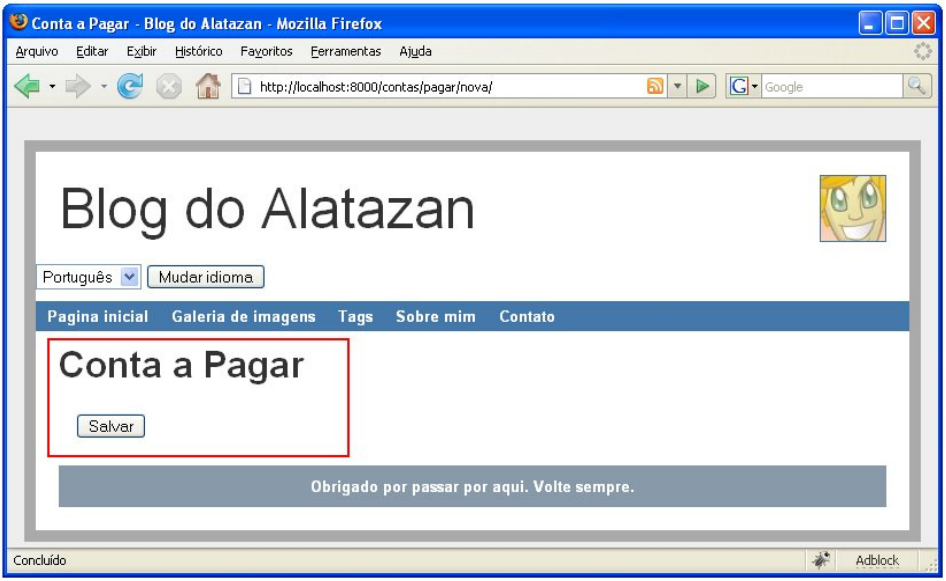
Esta é a view que deve atender às novas URLs que criamos. E para que ela funcione apropriadamente, vamos criar o template a que ela referencia. Para isso, crie o arquivo **"editar_conta.html"** na pasta **"contas/templates/contas"**, partindo da pasta do projeto, com o seguinte código dentro:

```
{% extends "base.html" %}


{% block titulo %}{{ titulo }} - {{ block.super }}{% endblock %}
{% block h1 %}{{ titulo }}{% endblock %}


{% block conteudo %}
<form method="post">
  <table class="form">


    <tr>
       <th> </th>
       <td><input type="submit" value="Salvar"/></td>
    </tr>
  </table>
</form>
{% endblock conteudo %}
```

Salve o arquivo e vá ao navegador nesta URL:

```
http://localhost:8000/contas/pagar/nova/
```

Veja como ela ficou:



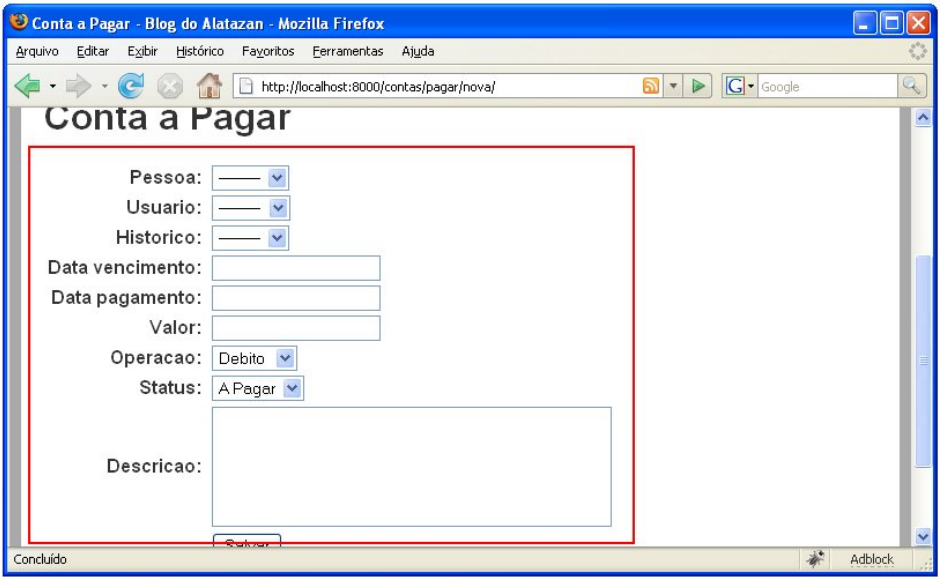
Humm... nada demais, certo? Mas voltando ao template que estamos criando, localize esta linha:

```
<table class="form">
```

E acrescente esta logo abaixo dela:

```
{{ form }}
```

Salve o arquivo. Feche o arquivo e veja a diferença no navegador:



Como você pode ver, a linha que acrescentamos fez **toda** a diferença. Isso é porque o **ModelForm** fez todo o trabalho de definir os campos, seus tipos e comportamentos, inclusive aqueles que são apenas uma caixa de seleção.

Acontece que ainda assim está longe do ideal, pois alguns campos ali deveriam ser exibidos de forma diferente.

Vamos portanto ajustar um pouco o formulário dinâmico para ficar ainda melhor. Para isso, abra o arquivo **"forms.py"** da pasta da aplicação **"contas"** para edição e localize esta linha de código:

```
        model = ContaPagar
```

Agora acrescente esta linha abaixo dela:

```
    exclude = ('usuario','operacao','data_pagamento')
```

Isso vai sumir com os campos **"usuario" "operacao"** e **"data_pagamento"** do formulário, e a partir de agora eles serão ignorados, assumindo seus valores *default*. Precisamos de fazer isso porque o usuário deve ser sempre o **usuário atual**, a **operação** de uma conta é definida quando ela é salva e já a **data de pagamento** deve ser atribuída quando o pagamento é feito.

Acrescente mais estas linhas abaixo dela:

```
    def __init__(self, *args, **kwargs):
        self.base_fields[
```

```
        'data_vencimento'
        ].widget = SelectDateWidget()


        super(FormContaPagar, self).__init__(*args, **kwargs)
```

A declaração deste método inicializador faz com que o campo **"data_vencimento"** passe a ter um *widget* de edição mais amigável.

Agora faça o mesmo com a classe **"FormContaReceber"**, acrescentando as seguintes linhas ao seu final:

```
        exclude = ('usuario','operacao','data_pagamento')


    def __init__(self, *args, **kwargs):
        self.base_fields[
        'data_vencimento'
        ].widget = SelectDateWidget()


        super(FormContaReceber, self).__init__(*args, **kwargs)
```

Agora vá ao início do arquivo e acrescente esta linha de código:

```
from django.forms.extras.widgets import SelectDateWidget
```

Salve o arquivo. Feche o arquivo.

Agora abra o arquivo **"views.py"** da mesma pasta, para edição, e localize este bloco de código:

```
def editar_conta(request, classe_form, titulo, conta_id=None):
    form = classe_form()


    return render_to_response(
        'contas/editar_conta.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Modifique-o para ficar assim:

```
def editar_conta(request, classe_form, titulo, conta_id=None):
    if request.method == 'POST':
```

```
        form = classe_form(request.POST)


        if form.is_valid():
            conta = form.save(commit=False)
            conta.usuario = request.user
            conta.save()


            return HttpResponseRedirect(conta.get_absolute_url())
    else:
        form = classe_form(
        initial={'data_vencimento': datetime.date.today()}
        )


    return render_to_response(
        'contas/editar_conta.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Veja que as novidades ficam por conta deste trecho:

```
        if form.is_valid():
            conta = form.save(commit=False)
            conta.usuario = request.user
            conta.save()
```

Primeiro de tudo, o formulário é **validado** através do método **"is_valid()"**. Se ele retornar **"True"** os dados são salvos mas o argumento **"commit=False"** determina que os dados não sejam persistidos no banco de dados. Isso porque logo em seguida o campo **"usuario"** deve receber o usuário autenticado do sistema (**"request.user"**) e por fim ele é salvo e persistido **de fato** no banco de dados com o método **"save()"** da **conta**.

A outra novidade está nesta linha de código:

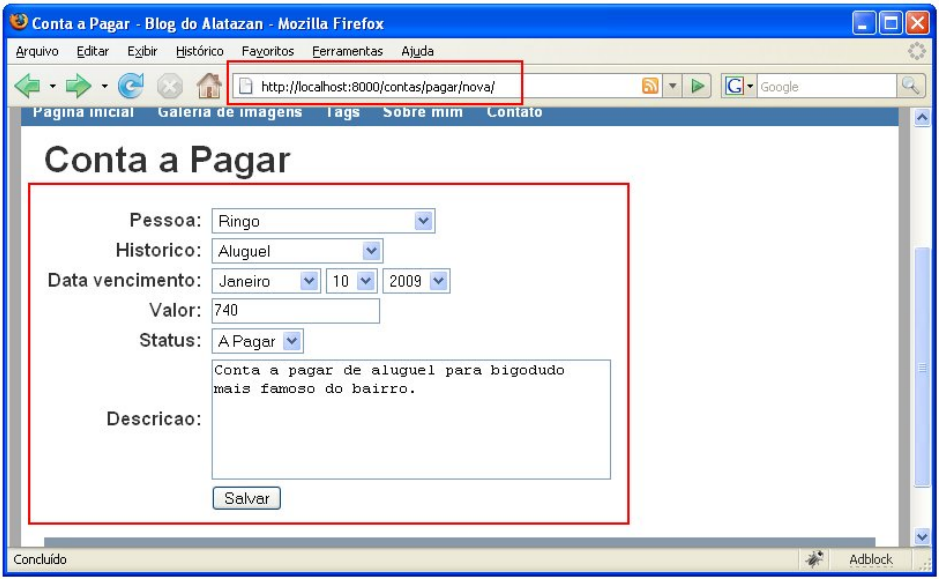
```
        form = classe_form(
        initial={'data_vencimento': datetime.date.today()}
        )
```

O argumento **"initial"** recebe os valores de inicialização do formulário, ou seja, seus valores **padrão**. Portanto o que a linha acima fez foi determinar que o valor de inicialização do campo **"data_vencimento"** deve ser a data atual.

Agora acrescente esta linha ao início do arquivo:

```
import datetime
```

Salve o arquivo. Feche o arquivo. Volte ao navegador e acrescente uma nova conta a pagar, assim por exemplo:



Ao final, clique sobre o botão **"Salvar"** para ver a conta ser salva no banco de dados e ser redirecionado à URL da conta.

Ajustando o formulário para edição de objetos

Nosso próximo passo agora é editar a conta. Para começar, precisamos de um link na página da conta para a página de edição da mesma. Então vamos editar o arquivo **"conta.html"** da pasta **"contas\templates\contas"** partindo da pasta do projeto e localizar esta linha:

```
{% block conteudo %}
```

Abaixo dela, acrescente esta:

```
<a href="editar/">{% trans "Editar" %}</a>
```

Salve o arquivo. Feche o arquivo. Volte ao navegador, atualize a página da conta e veja que temos um novo link:

Agora na pasta da aplicação **"contas"**, abra o arquivo **"urls.py"** para edição e acrescente estas duas novas URLs:

```
    url('^pagar/(?P<conta_id>\d+)/editar/$', 'editar_conta',
        {'classe_form': FormContaPagar,
         'titulo': 'Conta a Pagar'},
          name='editar_conta_a_pagar'),
    url('^receber/(?P<conta_id>\d+)/editar/$', 'editar_conta',
        {'classe_form': FormContaReceber,
         'titulo': 'Conta a Receber'},
          name='editar_conta_a_receber'),
```

Veja que fazemos referência a uma *view* que já criamos: **"editar_conta"**, mas desta vez nós vamos carregá-la para uma conta que já existe, indicada pelo argumento **"conta_id"**, que é citado na expressão regular da URL.

Salve o arquivo. Feche o arquivo. Agora abra o arquivo **"views.py"** da mesma pasta e localize esta linha de código:

```
def editar_conta(request, classe_form, titulo, conta_id=None):
```

Veja que na declaração da *view* o argumento **"conta_id"** é passado. Quando trata-se de uma **nova conta**, ele não recebe valor nenhum e assume seu valor padrão: **None**. Mas agora ele será atribuído com um valor: o ID da conta a editar.

Para fazer uso disso, precisamos acrescentar as seguintes linhas de código abaixo da linha que localizamos:

```
    if conta_id:
        conta = get_object_or_404(
        classe_form._meta.model,
        id=conta_id
        )
    else:
        conta = None
```

A variável **"classe_form"** traz a classe de formulário dinâmico indicado pela URL. Seu atributo **"_meta"** traz diversas informações de sua definição, e uma delas é o atributo **"model"**, que contém a classe de modelo à qual o formulário dinâmico está declarado. Em outras palavras, para a classe **"FormContaPagar"**, o elemento **"classe_form._meta.model"** contém **"ContaPagar"**, e para a classe **"FormContaReceber"** o mesmo elemento contém **"ContaReceber"**.

O que fizemos nesse trecho de código foi **carregar a conta** se o argumento **"conta_id"** tiver algum valor válido.

Agora localize esta linha:

```
        form = classe_form(request.POST)
```

E a modifique para ficar assim:

```
        form = classe_form(request.POST, instance=conta)
```

Localize esta linha também:

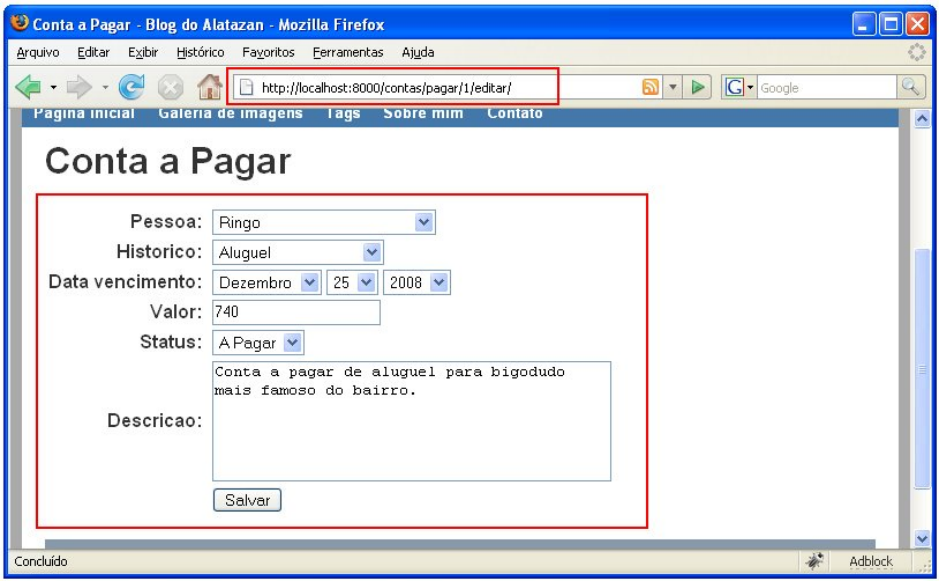
```
        form = classe_form(
        initial={'data_vencimento': datetime.date.today()}
        )
```

E a modifique para ficar assim:

```
        form = classe_form(
            initial={'data_vencimento': datetime.date.today()},
            instance=conta,
            )
```

Ambas as linhas que modificamos definem a mesma coisa: a variável **"conta"** é atribuída ao formulário dinâmico para edição. O formulário dinâmico se encarregará de verificar se a variável possui um valor válido e editar se for o caso.

Salve o arquivo. Feche o arquivo. Volte ao navegador, clique sobre o link **"Editar"** e veja a página que será carregada:



Viu como é simples?

Exclusão de objetos

Agora vamos criar um meio para excluir uma conta. Da mesma forma como fizemos com a edição, precisamos novamente de um link para a exclusão.

Para isso, abra para edição o arquivo **"conta.html"** da pasta **"contas\templates\contas"** e localize esta linha:

```
<a href="editar/">{% trans "Editar" %}</a>
```

Acrescente esta abaixo dela:

```
<a href="excluir/">{% trans "Excluir" %}</a>
```

Salve o arquivo. Feche o arquivo. Agora vamos declarar as URLs para tal.

Abra o arquivo **"urls.py"** da pasta da aplicação **"contas"** e acrescente estas novas URLs:

```
    url('^pagar/(?P<conta_id>\d+)/excluir/$', 'excluir_conta',
        {'classe': ContaPagar,
         'proxima': '/contas/pagar/'},
          name='excluir_conta_a_pagar'),
    url('^receber/(?P<conta_id>\d+)/excluir/$', 'excluir_conta',
        {'classe': ContaReceber,
         'proxima': '/contas/receber/'},
          name='excluir_conta_a_receber'),
```

Como pode ver, novamente fizemos algo muito semelhante às URLs que declaramos hoje, mas agora apontando para a *view* **"excluir_conta"**.

A novidade fica no argumento **"proxima"**, que aponta para uma URL. Este argumento será usado para redirecionar o navegador após a exclusão da conta, já que a exclusão não possui template ou página de destino.

Salve o arquivo. Feche o arquivo. Abra o arquivo **"views.py"** da mesma pasta para edição e acrescente este bloco de código ao final:

```
def excluir_conta(request, classe, conta_id, proxima='/contas/'):
    conta = get_object_or_404(classe, id=conta_id)
    conta.delete()


    request.user.message_set.create(
    message='Conta excluida com sucesso!'
    )


    return HttpResponseRedirect(proxima)
```

Humm... veja só:

As duas primeiras linhas são bastante conhecidas: a *view* recebe os argumentos que definimos na URL e logo a seguir a conta é carregada usando a função **"get_object_or_404()"**.

```
def excluir_conta(request, classe, conta_id, proxima='/contas/'):
    conta = get_object_or_404(classe, id=conta_id)
```

A primeira novidade está aqui:

```
    conta.delete()
```

Esta linha acima exclui a **conta** carregada do banco de dados, definitivamente.

E uma novidade mais diferente ainda é esta:

```
    request.user.message_set.create(
    message='Conta excluida com sucesso!'
    )
```

Este recurso só funciona quando há algum usuário autenticado. Ele tem a função de armazenar uma fila de mensagens para exibir para aquele usuário. Quando o usuário visualiza a mensagem, automaticamente ela é excluída do banco de dados.

A coisa bacana disso é que uma mensagem pode ficar armazenada até mesmo quando o usuário não está usando o site, por dias, semanas ou meses. Ela só será removida da fila quando o usuário visualizá-la!

Por fim, o navegador é redirecionado para o caminho indicado pela variável **"proxima"**:

```
    return HttpResponseRedirect(proxima)
```

Salve o arquivo. Feche o arquivo.

Agora para que o usuário possa visualizar a mensagem, abra o arquivo "base.html" da pasta "templates" do projeto e localize esta linha:

```
{% include "idiomas.html" %}
```

Acrescente estas linhas acima dela:

```
{% for msg in messages %}
<div class="mensagem">{{ msg }}</div>
{% endfor %}
```

Salve o arquivo. Feche o arquivo. Agora vamos dar um visual para essas mensagens. Abra o arquivo **"layout.css"** da pasta **"media"** do projeto e acrescente esta linha de código ao final:

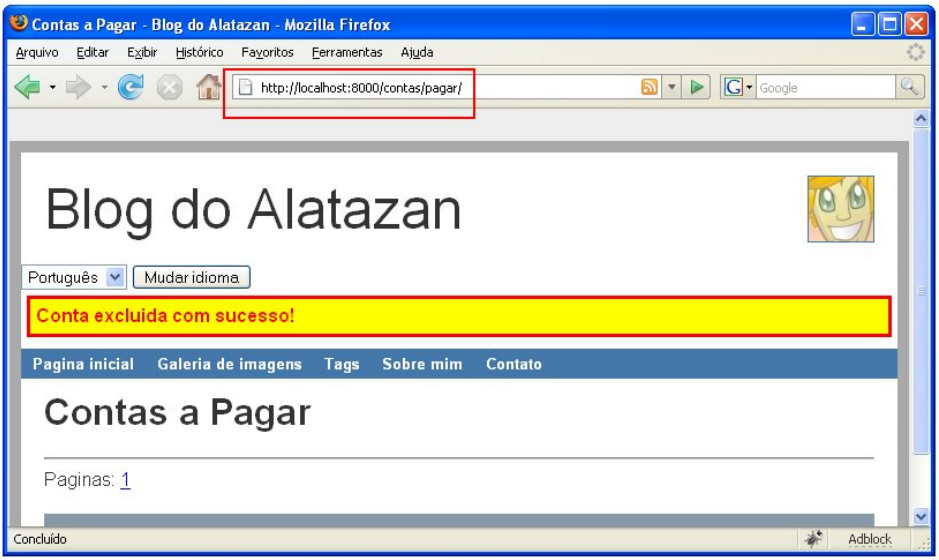
```
.mensagem {
  margin: 5px;
  padding: 5px;
  background: yellow;
  border: 3px solid red;
  font-weight: bold;
  color: red;
}
```

Salve o arquivo. Feche o arquivo. Agora volte ao navegador, na seguinte URL:

```
http://localhost:8000/contas/pagar/
```

Escolha uma conta a pagar já existente clicando sobre ela. Caso não exista uma, vá até a URL de Contas a Pagar do **Admin**, crie uma nova conta e após salvá-la, volte à URL de contas a pagar do site para ver a nova Conta a Pagar e clicar sobre ela.

Encontre o link **"Excluir"** e clique sobre ele, veja o que acontece:



O que achou? Isso não é bacana?

Agora que tal fazer o mesmo para as classes **"Pessoa"** e **"Historico"**? Legal! Mas isso agora é com você!

Registro de usuários, mudança de senha, essas coisas...

- Aí garoto! Como vai essa força aí?

A voz do outro lado da linha era inconfundível. Cartola vestia um sungão verde com borboletas psicodélicas azuis, pra combinar com a praia de Jericoacoara - essa dedução ele fez quando entendeu não havia nada mais psicodélico do que Jericoacoara... mas disso Alatazan não sabia, e o telefone ainda é só por voz...

- Fala folgado, já estou terminando a aplicação de contas, agora já está tudo

separadinho por usuários e amanhã eu vou fazer aquela coisarada toda de registro de usuário e tal...

- Bacana, camarada, e... *summing up*, o que você fez?
- Foi basicamente o seguinte:
  - A primeira coisa foi criar um campo **"usuario"** para as classes que eu queria separar por usuário;
  - Pra efetivar isso no banco de dados, usei a ferramenta **"manage.py reset"**, que exclui todas as tabelas e as cria novamente... **perdi todas** as minhas pessoas, históricos e contas, mas eu estava **consciente** disso;
  - Depois usei **ModelForms** pra fazer o cadastro de contas, ocultei alguns campos usando o atributo **"exclude"**, mudei o *widget* do campo de data...
  - Para cada uma das URLs de criação, edição e exclusão de contas, usei sempre parâmetros para definir qual era a classe em questão, assim evitei repetir código;
  - Na *view* de salvar contas, usei o método **"save()"** do formulário dinâmico, com o argumento **"commit=False"** pra dizer que não queria salvar no banco de dados ainda, pra dar tempo de informar qual era o usuário e só então efetivar a **persistência**;
  - Usei o argumento **"initial"** pra indicar um valor inicial ao formulário dinâmico;
  - E por fim, criei a função de excluir contas, usando o método **"delete()"**. Nessa hora o mais legal foi usar um recurso para jogar frases em uma **fila de mensagens** que é exibida ao usuário...
- Show de bola, meu irmão!
- Cartola, preciso dar uma volta por aí pra *jogar um grilo no esôfago*

Aquela expressão não era comum para Cartola, mas considerando que nem tudo na vida de Alatazan era comum a ele, ele deu de ombros, se despediu e voltou pra curtir a praia...

Agora, nosso próximo passo será o registro do usuário, mudança de senha e algumas definições de segurança para o que construímos hoje, afinal, ninguém quer ter suas contas sendo mexidas por qualquer um por aí né...

Capítulo 27: Funções de usuários...

Ainda na fila do banco, não faltava muito para Alatazan ser atendido. Um velho senhor japonês do tipo que tem bom gosto e gosta de fazer tudo ao seu jeito, aguardava calmamente a sua vez.

Sua senha era 2 posições antes da de Alatazan e ele ansiava por fazer seu registro e resolver suas coisas pessoais. Do nada os dois começaram a conversar...

- Meu nome é Geraldo Boaventura. É... eu sei que pareço ser alguém com um nome como "Turo Hayakawa" ou qualquer coisa assim, mas a verdade é que sou só metade japonês. Minha mãe era uma italiana que casou-se com um português, moramos por muito tempo no interior de Minas Gerais mas cansei daquelas aventuras. Hoje sou só um velho querendo resolver suas coisas da maneira mais simples possível...
- Quarenta e três!
- Senhor, acho que é a sua vez.
- Ahh, obrigado, obrigado.

Ele se virou para a moça e começaram o processo para a abrir sua conta bancária.

É... não era tão simples quanto ele gostaria... eram três senhas, dois conjuntos de combinações de letras e mais os números da agência, conta corrente, cartão de crédito, código de segurança... meu Deus!

Registro e outras funções para usuários

Se temos uma aplicação organizada e separada por usuários, o que falta são as funções para permitir que esses usuários se cadastrem no site, e iniciem sua vida por aqui, certo?

Então que tal começar pelo registro do usuário?

A primeira coisa é ajustar o menu principal para isso!

Abra o arquivo **"base.html"** da pasta **"templates"** do projeto para edição e localize esta linha de código:

```
<li><a href="{% url views.contato %}">Contato</a></li>
```

Agora acrescente estas linhas logo abaixo dela:

```
{% if user.is_authenticated %}
 <li><a href="{% url sair %}">{% trans "Sair" %}</a></li>
{% else %}
 <li>
 <a href="{% url entrar %}">{% trans "Entrar" %}</a>
 </li>
 <li>
 <a href="{% url registrar %}">{% trans "Registre-se" %}</a>
 </li>
{% endif %}
```

Como você pode ver, nós fazemos a seguinte verificação:

- Se há um usuário autenticado, ele tem uma nova opção: **Sair do site**, efetuando seu *logout*;
- Se o usuário não está autenticado, ele tem duas novas opções: **Entrar no site** (efetuar *logon*) ou **Registrar-se**.

Salve o arquivo. Feche o arquivo.

E agora vamos criar as URLs que indicamos no template. Para isso, abra o arquivo **"urls.py"** da pasta do projeto para edição, e acrescente as seguintes URLs:

```
(r'^entrar/$', 'django.contrib.auth.views.login',
    {'template_name': 'entrar.html'}, 'entrar'),
(r'^sair/$', 'django.contrib.auth.views.logout',
    {'template_name': 'sair.html'}, 'sair'),
(r'^registrar/$', 'views.registrar', {}, 'registrar'),
```

Observe que as duas primeiras URLs (**"entrar"** e **"sair"**) referem-se a duas *views* já existentes, que fazem parte da aplicação **"auth"**. Elas são duas das **generic views** que já fazem a maior parte do trabalho, dependendo somente por definir seus templates para que elas funcionem adequadamente.

Já a terceira URL (**"registrar"**) faz referência a uma nova *view*, que vamos criar agora mesmo!

Salve o arquivo. Feche o arquivo. Abra o arquivo **"views.py"** da pasta do projeto e acrescente as seguintes linhas ao final:

```
def registrar(request):
    if request.method == 'POST':
        form = FormRegistro(request.POST)


        if form.is_valid():
            novo_usuario = form.save()
            return HttpResponseRedirect('/')
    else:
        form = FormRegistro()


    return render_to_response(
        'registrar.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

A nova *view* não apresenta grandes novidades. Apenas fazemos referência a um novo formulário dinâmico e o uso de um novo template.

Antes de partir para o novo formulário e o novo template, precisamos importar os dois elementos estranhos a este arquivo. Portanto, vá até o início do arquivo e acrescente estas duas novas linhas de código:

```
from django.http import HttpResponseRedirect


from forms import FormRegistro, FormContato
```

Salve o arquivo.

Agora, ainda na pasta do projeto, crie o novo arquivo **"forms.py"** com o seguinte código dentro:

```
from django.contrib.auth.models import User
from django import forms
from django.utils.translation import ugettext as _


class FormRegistro(forms.ModelForm):
```

```
    class Meta:
        model = User
```

Faça ainda uma coisa a mais: volte ao arquivo **"views.py"** que acabamos de modificar e localize este bloco de código:

```
class FormContato(forms.Form):
    nome = forms.CharField(max_length=50, label=_('Nome'))
    email = forms.EmailField(required=False, label=_('E-mail'))
    mensagem = forms.Field(
    widget=forms.Textarea, label=_('Mensagem')
    )

    def enviar(self):
        titulo = 'Mensagem enviada pelo site'
        destino = 'alatazan@gmail.com'
        texto = """
        Nome: %(nome)s
        E-mail: %(email)s
        Mensagem:
        %(mensagem)s
        """ % self.cleaned_data

        send_mail(
            subject=titulo,
            message=texto,
            from_email=destino,
            recipient_list=[destino],
            )
```

Recorte e cole o bloco acima no arquivo **"forms.py"** que acabamos de criar. Faça o mesmo com a seguinte linha, que deve colada entre as primeiras linhas do novo arquivo:

```
from django.core.mail import send_mail
```

Assim o nosso código fica um pouco mais organizado.

Salve ambos os arquivos, e feche-os. Agora precisamos criar o novo template,

chamado **"registrar.html"** na pasta **"templates"** da pasta do projeto, com o seguinte código dentro:

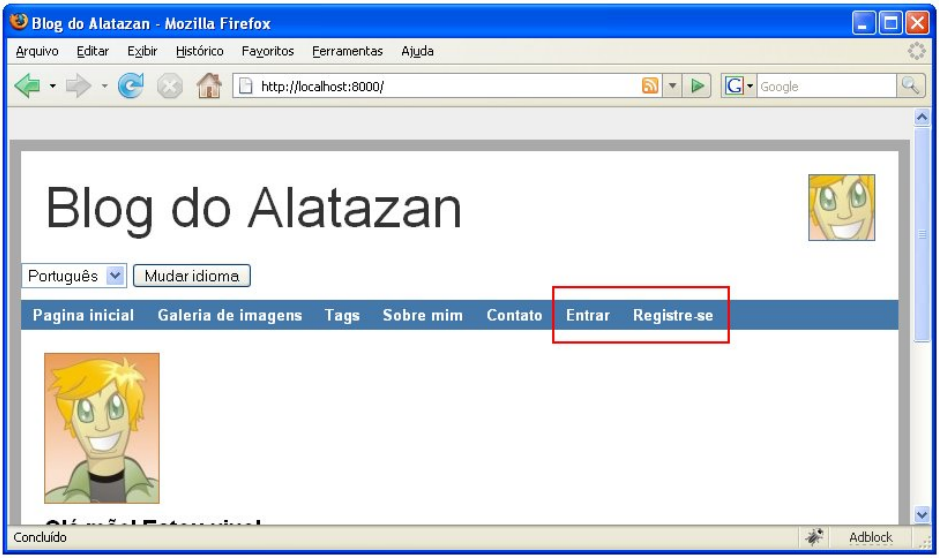
```
{% extends "base.html" %}


{% load i18n %}


{% block titulo %}{% trans "Registro de usuario" %} -
{{ block.super }}{% endblock %}
{% block h1 %}{% trans "Registro de usuario" %}{% endblock %}


{% block conteudo %}{{ block.super }}


<form method="post">
 <table class="form">
  {{ form }}


  <tr>
   <th> </th>
   <td>
   <input type="submit" value="{% trans "Registrar" %}"/>
   </td>
  </tr>
 </table>
</form>
{% endblock conteudo %}
```

Salve o arquivo. Feche o arquivo. Observe que continuamos fazendo aquilo que já conhecemos: estendemos o template **"base.html"** e seus **blocos**, e acrescentamos um novo formulário ali dentro.

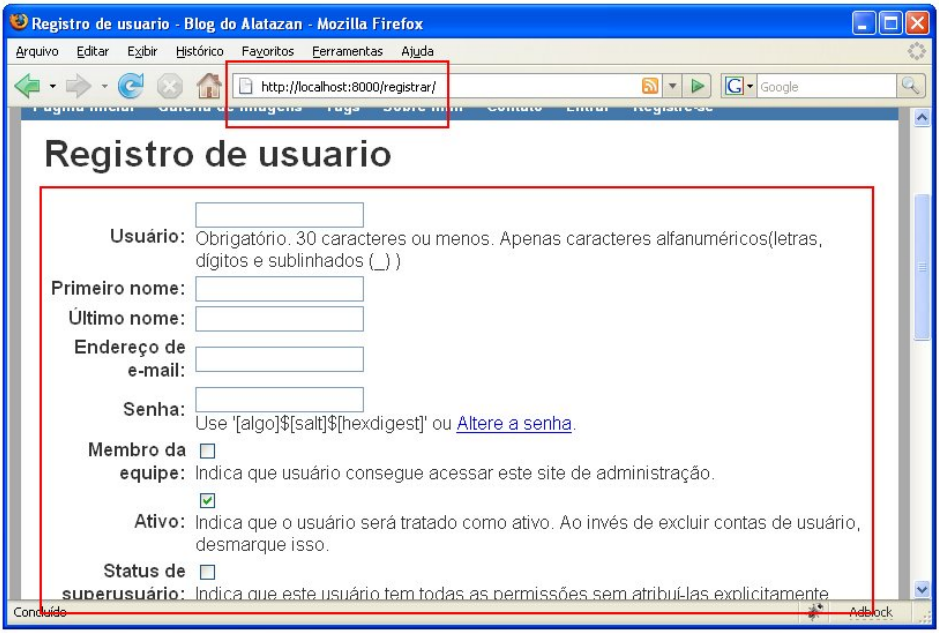
Agora execute seu projeto, clicando duas vezes sobre o arquivo **"executar.bat"**, e abra o navegador na URL principal:

```
http://localhost:8000/
```

Certifique-se de que você não está autenticado pelo **Admin** e seu resultado deve ser este:



E ao clicar no *link* **"Registrar-se"**, o resultado deve ser este:



Opa! Mas pera lá... a dose foi um pouco mais forte do que precisamos!

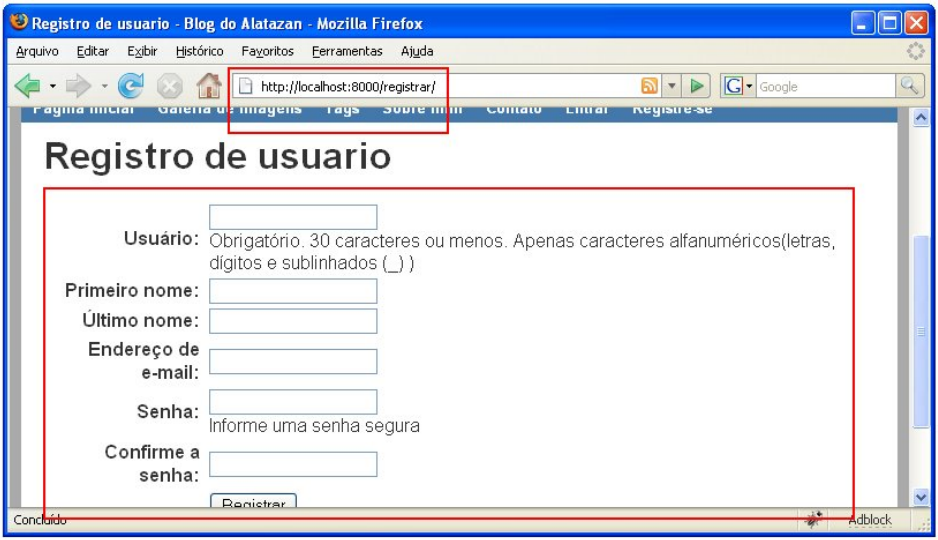
Então agora volte ao arquivo **"forms.py"** da pasta do projeto para edição e localize este bloco de código:

```
class FormRegistro(forms.ModelForm):
    class Meta:
        model = User
```

Acrescente mais estas linhas de código logo abaixo do bloco localizado:

```
        fields = (
        'username','first_name','last_name','email','password'
        )


    confirme_a_senha = forms.CharField(
        max_length=30, widget=forms.PasswordInput
        )


    def __init__(self, *args, **kwargs):
        self.base_fields[
        'password'
        ].help_text = 'Informe uma senha segura'
        self.base_fields[
        'password'
        ].widget = forms.PasswordInput()
        super(FormRegistro, self).__init__(*args, **kwargs)
```

Salve o arquivo. Volte ao navegador e atualize a página com **F5**. Veja como ficou:



Uau! Melhorou muito! Mas ainda faltam algumas coisas para ter seu pleno funcionamento.

Para permitir o registro do novo usuário, algumas validações são necessárias:

- Verificar se já existe algum usuário com aquele **"username"**;
- Verificar se já existe algum usuário com aquele **"e-mail"**;
- Verificar se a senha informada foi **confirmada** corretamente;
- e criptografar a senha antes de salvar.

Não se preocupe, isso não é complicado. Para fazer isso, volte ao arquivo **"forms.py"** para edição, e acrescente este bloco de código ao final da classe **"FormRegistro"** (logo abaixo da linha que modificamos por último):

```
    def clean_username(self):
        if User.objects.filter(
            username=self.cleaned_data['username'],
            ).count():
            raise forms.ValidationError(
            'Ja existe um usuario com este username'
            )
```

```
        return self.cleaned_data['username']

    def clean_email(self):
        if User.objects.filter(
            email=self.cleaned_data['email'],
            ).count():
            raise forms.ValidationError(
            'Ja existe um usuario com este e-mail'
            )

        return self.cleaned_data['email']

    def clean_confirme_a_senha(self):
        if self.cleaned_data[
        'confirme_a_senha'
        ] != self.data['password']:
            raise forms.ValidationError(
            'Confirmacao da senha nao confere!'
            )

        return self.cleaned_data['confirme_a_senha']

    def save(self, commit=True):
        usuario = super(FormRegistro, self).save(commit=False)

        usuario.set_password(self.cleaned_data['password'])

        if commit:
            usuario.save()

        return usuario
```

Você pode notar que são **quatro métodos**, que fazem respectivamente o que citamos anteriormente:

O primeiro método valida se já existe um usuário com o **"username"** informado. Para isso, ele faz uso do método **"count()"** da queryset, que retorna o total de registros da classe **"User"** para o filtro onde **"username"** seja o que o usuário informou (**"self.cleaned_data['username']"**):

```
    def clean_username(self):
        if User.objects.filter(
            username=self.cleaned_data['username'],
            ).count():
            raise forms.ValidationError(
            'Ja existe um usuario com este username'
            )


        return self.cleaned_data['username']
```

Vale ressaltar que para cada campo do formulário, é possível declarar um método **"clean_NOMEDOCAMPO(self)"** para validar aquele campo isoladamente.

Ao levantar a exceção **"forms.ValidationError()"**, dizemos ao Django que aquele campo não foi informado corretamente e informamos a mensagem que explica o porquê disso.

Em seguida fazemos o mesmo, mas desta vez a nossa atenção é dada ao campo **"email"**:

```
    def clean_email(self):
        if User.objects.filter(
            email=self.cleaned_data['email'],
            ).count():
            raise forms.ValidationError(
            'Ja existe um usuario com este e-mail'
            )


        return self.cleaned_data['email']
```

O método seguinte já possui um artifício um pouco mais delicado: comparar os dois campos informados de senha (**"password"**) e confirmação da senha (**"confirme_a_senha"**):

```
    def clean_confirme_a_senha(self):
```

```
        if self.cleaned_data[
        'confirme_a_senha'
        ] != self.data['password']:
            raise forms.ValidationError(
            'Confirmacao da senha nao confere!'
            )

        return self.cleaned_data['confirme_a_senha']
```

Ali há uma pequena armadilha. Note que a nossa condição compara um item do dicionário **"self.cleaned_data"** com um item do dicionário **"self.data"**. Isso é necessário pois o método de validação do campo não faz parte de uma sequência definida.

A sequência das validações é feita aleatoriamente, dependendo de como a memória posicionou os campos, e portanto, não é seguro verificar o valor de outro campo (**"password"**, no caso) no dicionário de valores **já validados** (**"self.cleaned_data"** neste caso) simplesmente porque ele pode não ter sido validado ainda. Mas o dicionário de **valores informados** (**"self.data"**) é uma fonte segura dessa informação.

No caso do campo **"confirme_a_senha"**, podemos confiar pois se trata do campo que está sendo validado neste método, e portanto, é certo que uma validação básica foi feita antes disso e ele seguramente estará no dicionário **"self.cleaned_data"**.

Por fim, sobrepusemos o método **"save()"** para repetir o seu comportamento (com a função **"super()"**) mas interferir em uma coisa: atribuir a senha usando o método que a criptografa para a sua atribuição. E logo em seguida, salvamos o registro (se assim estiver definido pelo argumento **"commit"**):

```
    def save(self, commit=True):
        usuario = super(FormRegistro, self).save(commit=False)

        usuario.set_password(self.cleaned_data['password'])

        if commit:
            usuario.save()

        return usuario
```

Salve o arquivo. Feche o arquivo.

Agora você pode registrar seu usuário numa boa.

Funções de logon e logout

Você se lembra de que nós criamos duas URLs usando *generic views* para **entrar** e **sair** do site, certo? Pois bem, precisamos agora criar os templates para elas.

Na pasta **"templates"** do projeto, crie o arquivo **"entrar.html"** com o seguinte código dentro:

```
{% extends "base.html" %}


{% load i18n %}


{% block titulo %}{% trans "Entrar" %} -
{{ block.super }}{% endblock %}
{% block h1 %}{% trans "Entrar" %}{% endblock %}


{% block conteudo %}{{ block.super }}


<form method="post">
 <table class="form">
  {{ form }}


  <tr>
   <th> </th>
   <td><input type="submit" value="{% trans "Entrar" %}"/></td>
  </tr>
 </table>
</form>
{% endblock conteudo %}
```

Notou que é um template muito parecido com outros que já fizemos, especialmente o que criamos para o registro do usuário? Sim, sim... eles são muito semelhantes!

A *generic view* já se trata de colocar ali os campos corretamente, basta usar a variável **{{ form }}** dentro de uma tag HTML **<FORM>**.

Salve o arquivo. Feche o arquivo. Agora crie mais um arquivo na mesma pasta, chamado **"sair.html"**, com o seguinte código dentro:

```
{% extends "base.html" %}


{% load i18n %}


{% block titulo %}{% trans "Voce saiu do sistema" %} -
{{ block.super }}{% endblock %}
{% block h1 %}{% trans "Voce saiu do sistema" %}{% endblock %}


{% block conteudo %}{{ block.super }}


Voce saiu do sistema. Obrigado e volte sempre!
{% endblock conteudo %}
```

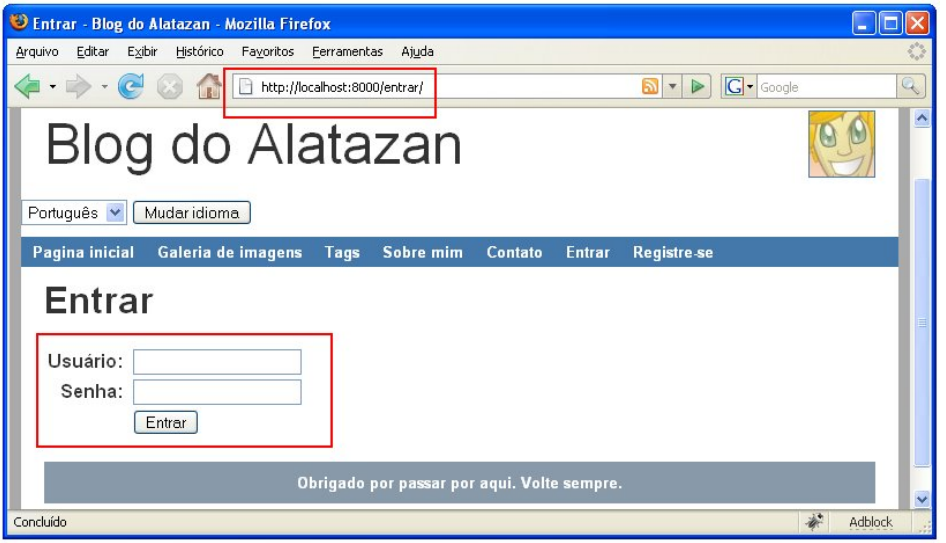
Templatezinho básico, só pra avisar o usuário de que ele saiu do sistema com êxito e pode voltar assim que quiser.

Salve o arquivo. Feche o arquivo.

Agora no navegador, carregue a seguinte URL:

```
http://localhost:8000/entrar/
```

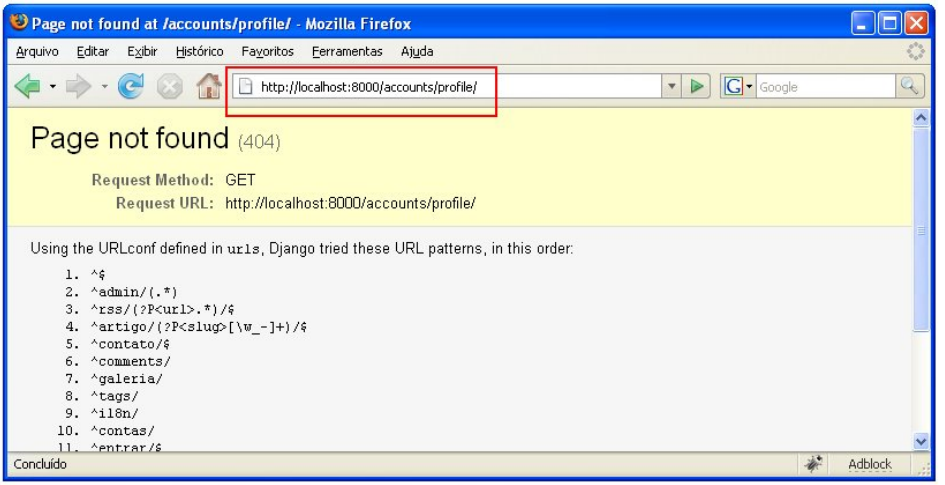
Veja como ela é mostrada:



Legal! Informe ali um usuário e senha. Pode ser algum usuário que você tenha criado usando a página **"Registre-se"** ou pode ser o que temos trabalhado desde o começo do estudo:

- Usuário: **"admin"**
- Senha: **"1"**

Clique no botão **"Entrar"** e veja o efeito:



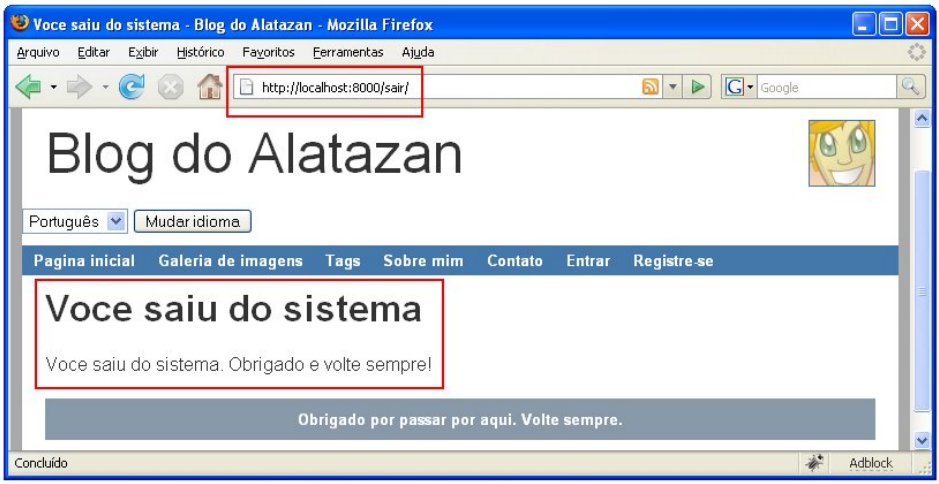
O que acontece é que, por padrão o Django entende que seu usuário terá uma página própria e que esta página própria por padrão estaria na URL **"/accounts/profile/"**. Como não é o nosso caso, vamos resolver isso assim:

Vá até a pasta do projeto e abra o arquivo **"settings.py"** para edição. Acrescente a seguinte linha de código ao final do arquivo:

```
LOGIN_REDIRECT_URL = '/contas/'
```

Salve o arquivo. Feche o arquivo. Volte à URL para **entrar no site** novamente e veja que dessa vez a página de **Contas** é mostrada!

Feito isso, clique sobre o link **"Sair"** e veja que a página é carregada assim:



Fechando views para aceitarem somente o acesso de usuários autenticados

Agora vá até a pasta da aplicação **"contas"** e abra o arquivo **"views.py"** para edição. Localize esta linha de código:

```
from django.core.paginator import Paginator
```

Acrescente esta outra linha logo abaixo dela:

```
from django.contrib.auth.decorators import login_required
```

"login_required" é o *decorator* que obriga o usuário a se autenticar ao acessar uma *view*.

Agora localize esta outra linha:

```
def contas(request):
```

Acrescente esta acima dela:

```
@login_required
```

Localize esta:

```
def conta(request, conta_id, classe):
```

Acrescente novamente aquela mesma linha de código aqui, logo acima dela:

```
@login_required
```

Localize esta:

```
def conta_pagamento(request, conta_id, classe):
```

Faça o mesmo:

```
@login_required
```

Localize esta:

```
def contas_por_classe(request, classe, titulo):
```

Novamente:

```
@login_required
```

Localize esta:

```
def editar_conta(request, classe_form, titulo, conta_id=None):
```

Outra vez:

```
@login_required
```

Localize esta:

```
def excluir_conta(request, classe, conta_id, proxima='/contas/'):
```

E agora mais outra vez:

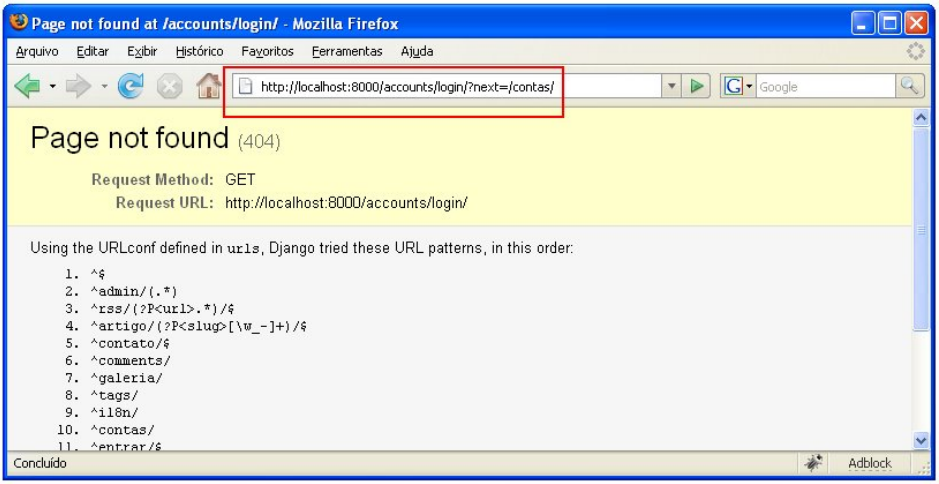
```
@login_required
```

Ufa! É isso aí.

Salve o arquivo. Feche o arquivo. Agora volte ao navegador e tente acessar esta URL:

```
http://localhost:8000/contas/
```

E veja o que acontece:



Veja que a nossa URL de **Contas** agora está inacessível, pois para ser isso, é necessário que o usuário se autentique. Só que o Django tentou abrir uma URL padrão para a autenticação do usuário, e esta URL não existe.

Para resolver isso, abra o arquivo **"settings.py"** da pasta do projeto e acrescente esta linha ao final:

```
LOGIN_URL = '/entrar/'
```

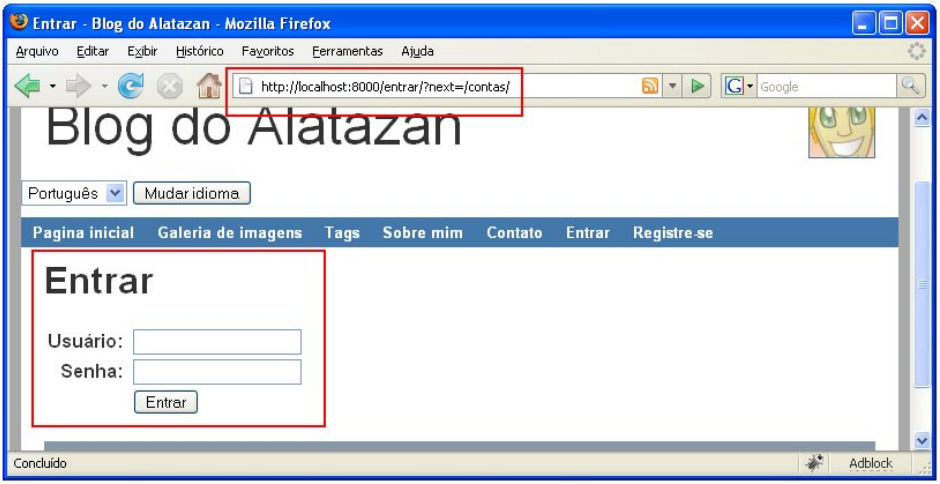
Aproveite e acrescente logo esta também:

```
LOGOUT_URL = '/sair/'
```

Salve o arquivo. Feche o arquivo. Agora tente novamente a URL no navegador:

```
http://localhost:8000/contas/
```

E o que vemos já é bem mais satisfatório:



Observe que ao informar seu usuário e senha para se autenticar, ao clicar sobre o botão **"Entrar"** a URL requisitada é carregada de forma direta!

Organizando a segurança das views

Outra coisa importante agora é ajustar algumas coisinhas nas nossas *views* para que elas não sejam vulneráveis. Imagine por exemplo um usuário excluindo a conta de outro usuário! Isso é inadmissível, não é?

Pois então vá até a pasta da aplicação **"contas"** e abra o arquivo **"views.py"** para edição. Localize este trecho de código:

```
def conta(request, conta_id, classe):
    conta = get_object_or_404(classe, id=conta_id)
```

Acrescente logo abaixo dele:

```
    if conta.usuario != request.user:
        raise Http404
```

Este código faz com que, se o usuário autenticado não for o dono desta conta, uma página de **erro 404** é mostrada. Para que ela funcione corretamente localize esta linha no início do arquivo:

```
from django.http import HttpResponseRedirect
```

E a modifique para ficar assim:

```
from django.http import HttpResponseRedirect, Http404
```

Agora localize este outro trecho de código:

```
def conta_pagamento(request, conta_id, classe):
    conta = get_object_or_404(classe, id=conta_id)
```

Acrescente abaixo dele:

```
    if conta.usuario != request.user:
        raise Http404
```

Localize mais este trecho:

```
def editar_conta(request, classe_form, titulo, conta_id=None):
    if conta_id:
        conta = get_object_or_404(
        classe_form._meta.model,
        id=conta_id
        )
```

Acrescente abaixo dele (tenha cuidado para que o código inserido fique dentro do bloco do **"if"**):

```
        if conta.usuario != request.user:
            raise Http404
```

Localize este trecho também:

```
def excluir_conta(request, classe, conta_id, proxima='/contas/'):
    conta = get_object_or_404(classe, id=conta_id)
```

E acrescente este logo abaixo:

```
    if conta.usuario != request.user:
        raise Http404
```

Assim, todas as *views* que indicam uma conta específica estão protegidas. Mas só isso ainda não é suficiente. Precisamos também filtrar outras duas *views* para mostrar somente as contas do usuário atual, não é?

Localize este trecho de código:

```
    contas_a_pagar = ContaPagar.objects.filter(
        status=CONTA_STATUS_APAGAR,
        )
    contas_a_receber = ContaReceber.objects.filter(
        status=CONTA_STATUS_APAGAR,
```

```
        )
```

Modifique-o para ficar assim:

```
    contas_a_pagar = ContaPagar.objects.filter(
        status=CONTA_STATUS_APAGAR, usuario=request.user,
        )
    contas_a_receber = ContaReceber.objects.filter(
        status=CONTA_STATUS_APAGAR, usuario=request.user,
        )
```

Observe que acrescentamos mais um elemento ao filtro: **"usuario=request.user,"**.

Localize este trecho também:

```
def contas_por_classe(request, classe, titulo):
    contas = classe.objects.order_by('status','data_vencimento')
```

Modifique para ficar assim:

```
def contas_por_classe(request, classe, titulo):
    contas = classe.objects.filter(
        usuario=request.user
        ).order_by('status','data_vencimento')
```

Novamente, trabalhamos no filtro pelo campo **"usuario"**.

Agora salve o arquivo. Feche o arquivo.

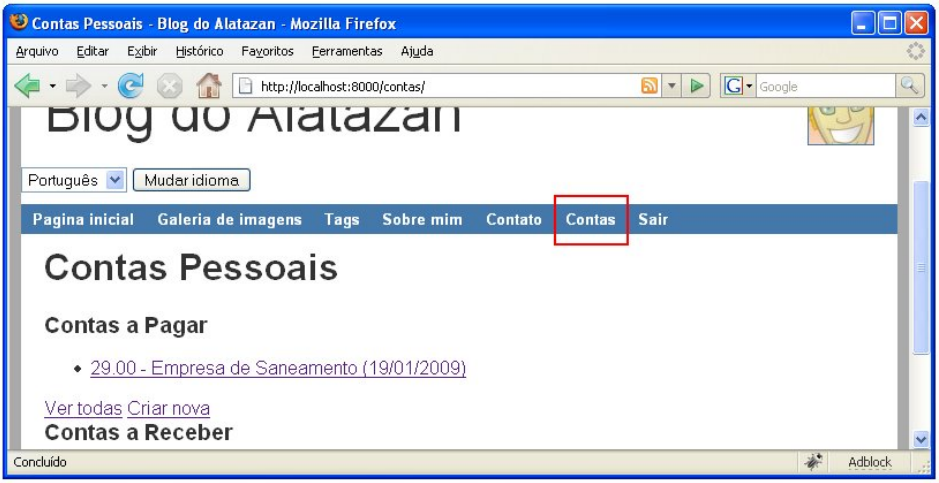
Precisamos fazer uma última mudança, coisa antiga... Abra o arquivo **"base.html"** da pasta **"templates"** do projeto para edição e localize esta linha de código:

```
     <li><a href="{% url views.contato %}">Contato</a></li>
```

E acrescente esta linha logo abaixo dela:

```
     <li><a href="{% url contas %}">Contas</a></li>
```

Salve o arquivo. Feche o arquivo. Veja no navegador como as nossas páginas estão:



Criando uma view com acesso especial

Agora vamos saber um pouco sobre **permissões de acesso**. Elas serão úteis na nova *view* que vamos criar agora, para listar todos os usuários do sistema. Isso será acessível somente para quem estiver autenticado e tiver acesso especial para isso.

Na pasta do projeto, abra o arquivo **"urls.py"** para edição e acrescente a seguinte URL:

```
    (r'^todos_os_usuarios/$', 'views.todos_os_usuarios',
        {}, 'todos_os_usuarios'),
```

Salve o arquivo. Feche o arquivo.

Ainda na mesma pasta, abra o arquivo **"views.py"** para edição e acrescente as seguintes linhas de código ao final:

```
@permission_required('ver_todos_os_usuarios')
def todos_os_usuarios(request):
    usuarios = User.objects.all()
    return render_to_response(
        'todos_os_usuarios.html',
        locals(),
        context_instance=RequestContext(request),
        )
```

Viu que trabalhamos uma coisa diferente? Esta:

```
@permission_required('contas.ver_todos_os_usuarios')
```

Este *decorator* determina que somente usuários que tenham a permissão **"ver_todos_os_usuarios"** da aplicação **"contas"** podem acessar a *view* em questão.

Vá agora ao início do arquivo e acrescente estas linhas de código:

```
from django.contrib.auth.models import User
from django.contrib.auth.decorators import permission_required
```

Salve o arquivo. Feche o arquivo.

Agora vamos criar o template da *view* que acabamos de criar. Na pasta **"templates"** do projeto, crie um novo arquivo chamado **"todos_os_usuarios.html"** com o seguinte código dentro:

```
{% extends "base.html" %}


{% load i18n %}


{% block titulo %}{% trans "Todos os usuarios do sistema" %} -
{{ block.super }}{% endblock %}
{% block h1 %}{% trans "Todos os usuarios do sistema" %}
{% endblock %}


{% block conteudo %}{{ block.super }}


<ul>
 {% for usuario in usuarios %}
 <li>{{ usuario }}</li>
 {% endfor %}
</ul>
{% endblock conteudo %}
```

Salve o arquivo. Feche o arquivo. Agora na pasta da aplicação **"contas"**, abra o arquivo **"models.py"** para edição e localize o seguinte trecho de código:

```
class Conta(models.Model):
    class Meta:
```

```
        ordering = ('-data_vencimento', 'valor')
```

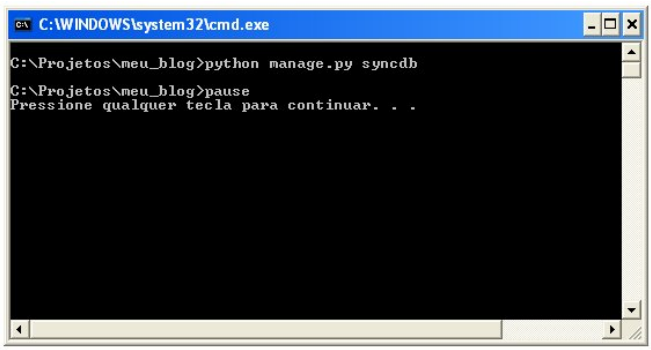
Acrescente este bloco de código logo abaixo:

```
    permissions = (
        ('ver_todos_os_usuarios', 'Ver todos os usuarios'),
        )
```

Isso vai criar uma nova permissão: **"Ver todos os usuarios"**, sob o codinome **"ver_todos_os_usuarios"**.

Salve o arquivo. Feche o arquivo.

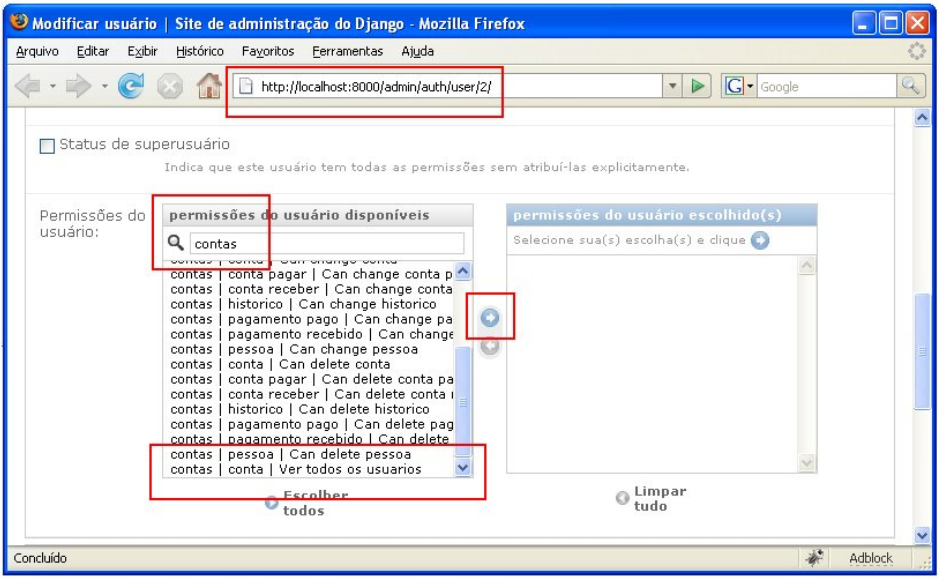
Agora falta uma última coisa: atualizar a geração do banco de dados para ter a nova permissão. Para isso, clique duas vezes sobre o arquivo **"gerar_banco_de_dados.bat"** da pasta do projeto e o resultado deve ser este:



Não há mensagem esclarecendo o que houve, mas de fato, a permissão é gerada nesse momento. Agora vá ao navegador e carregue a URL para edição de um usuário no **Admin**, como esta por exemplo:

```
http://localhost:8000/admin/auth/user/2/
```

Veja como a página é mostrada:



Note que o último item é a permissão que criamos: **"contas | conta | Ver todos os usuarios"**. Clique sobre ele para selecioná-lo e clique sobre o **ícone azul com uma setinha** à direita para movê-lo para a caixa da direita. Feito isso, salve o usuário, e ele terá essa permissão especial!

Agora você pode se autenticar com o usuário que tem tal permissão e carregar a URL protegida, pois ele terá acesso. Do contrário, a página **"Entrar"** será carregada solicitando a autenticação de um usuário que tenha a permissão necessária.

Usuários marcados como **"Superusuário"** não são barrados por permissões, pois possuem todas elas atribuídas por definição, mesmo que não o estejam no cadastro do usuário.

Alteração de senha

Agora, que tal usar um pouco mais das *generic views* da aplicação de **autenticação**? Então vamos escolher a alteração de senha para brincar um pouco.

Na pasta do projeto, abra para edição o arquivo **"urls.py"** e acrescente as seguintes URL:

```
    (r'^mudar_senha/$',
    'django.contrib.auth.views.password_change',
```

```
    {'template_name': 'mudar_senha.html'},
    'mudar_senha'),
    (r'^mudar_senha/concluido/$',
    'django.contrib.auth.views.password_change_done',
    {'template_name': 'mudar_senha_concluido.html'},
    'mudar_senha_concluido'),
```

Na primeira URL, fazemos referência à *generic view* **"django.contrib.auth.views.password_change"**, demos o nome à URL de **"mudar_senha"** e usamos o template **"mudar_senha.html"**.

A segunda URL é necessária e trata-se daquela que será chamada após a alteração da senha. Ela faz uso da *generic view* **"django.contrib.auth.views.password_change_done"**

Salve o arquivo. Feche o arquivo. Agora na pasta **"templates"** do projeto, crie um novo arquivo chamado **"mudar_senha.html"** com o seguinte código dentro:

```
{% extends "base.html" %}

{% load i18n %}

{% block titulo %}{% trans "Mudar senha" %} -
{{ block.super }}{% endblock %}
{% block h1 %}{% trans "Mudar senha" %}{% endblock %}

{% block conteudo %}{{ block.super }}

<form method="post">
 <table class="form">
  {{ form }}

  <tr>
   <th> </th>
   <td>
   <input type="submit" value="{% trans "Mudar senha" %}"/>
   </td>
```

```
  </tr>
 </table>
</form>
{% endblock conteudo %}
```

Nada demais, né?

Salve o arquivo. Feche o arquivo.

Agora crie outro arquivo, chamado **"mudar_senha_concluido.html"** na mesma pasta, com o seguinte código dentro:

```
{% extends "base.html" %}


{% load i18n %}


{% block titulo %}{% trans "Senha alterada" %} -
{{ block.super }}{% endblock %}
{% block h1 %}{% trans "Senha alterada" %}{% endblock %}


{% block conteudo %}{{ block.super }}


<b>A sua senha foi alterada com sucesso!</b>


{% endblock conteudo %}
```

Nada demais também.

Salve o arquivo. Feche o arquivo. Agora só precisamos de acrescentar essa opção ao menu e para isso só é preciso abrir o arquivo **"base.html"** da mesma pasta de templates para edição e localizar esta linha de código:

```
    <li><a href="{% url sair %}">{% trans "Sair" %}</a></li>
```

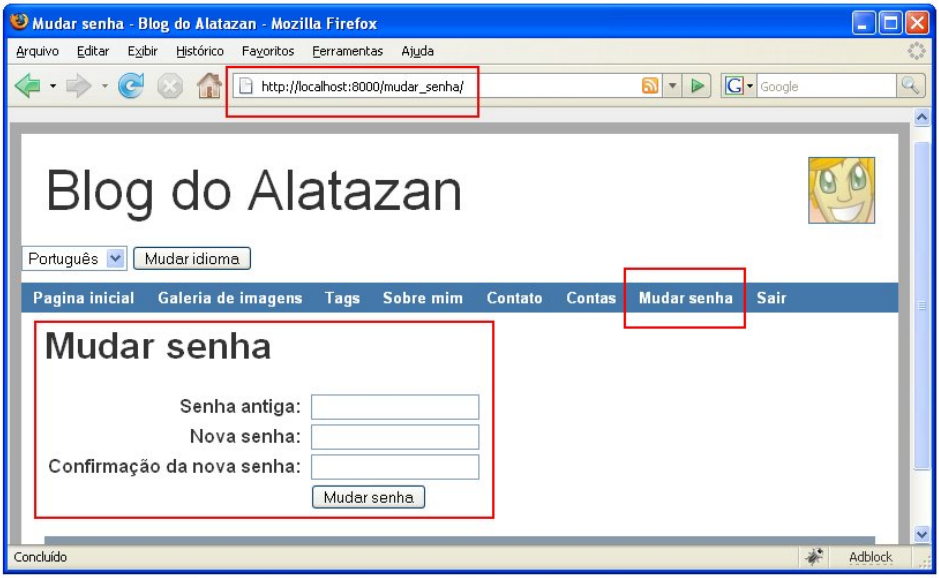
Acrescente esta outra linha de código acima dela:

```
    <li>
    <a href="{% url mudar_senha %}">{% trans "Mudar senha" %}</a>
    </li>
```

Salve o arquivo. Feche o arquivo. Agora volte ao navegador, na seguinte URL:

```
http://localhost:8000/mudar_senha/
```

E veja a página que é carregada:



E aí, melzinho na chupeta, não é mesmo?

E agora, que tal um pouco mais de testes?

- Visceral! Animal!

Foi a reação de Cartola, que ainda não conhecia bem as *generic views* da aplicação **"auth"**.

- Cara, isso é muito maneiro!

A cara de Alatazan passou do amarelo habitual para amarelinho, amarelinhozinho, quase-branco-amarelado, branco-de-fato e tornou ao mesmo amarelo habitual fazendo o caminho de volta passando pelo quase-branco-amarelado, amarelinhozinho e amarelinho.

É que a associação entre **"vísceras"** e algo realmente **"animal"** era bastante chato do ponto de vista de um estômago um pouco sensível. Mas logo ele se tocou que *"visceral"* e *"animal"* são expressões mais *cool* e com um significado bastante simpático, por sinal.

- Olha só, com exceção da *view* de **registro de usuário**, o que fizemos aqui foi usar bastante as *generic views*, então vamos dar uma passada rápida no que

fizemos, ok?

- O registro do usuário foi feito usando **ModelForms**, e pra isso nós fizemos uso de métodos de validação do tipo **"clean_NOME_DO_CAMPO"** e também uma ideia bem sacada foi sobrepôr o método **"save()"** e forçar a criptografia da senha com **"set_password()"**. Muito legal isso!

- Inclusive a validação é feita usando uma exceção, que ao invés de derrubar todo o site, apenas exibe a mensagem da validação. Ela se chama **"forms.ValidationError()"**;

- Movemos o formulário de contato para o novo arquivo **"forms.py"** pra dar uma organizada por ali;

- No mais, foram *generic views*.

- As duas primeiras que usamos foram as de **"login"** e **"logout"**, definimos templates para elas e babau!

- Depois definimos algumas *settings* pra trabalhar bem com essas *views*, como por exemplo a **"LOGIN_REDIRECT_URL"** e a **"LOGIN_URL"**;

- Fizemos uso também dos decorators **"@login_required"** e **"@permission_required"**. O primeiro protege a *view* para ser acessada somente por quem está autenticado. Já o segundo garante que a *view* só será acessada por quem tenha uma permissão especial;

- E falando em permissões, declaramos o atributo **"permissions"** da classe **"Meta"** de uma classe de modelo para criar essa permissão especial, atualizamos o banco de dados e liberamos a permissão para um usuário usando o **Admin**;

- Também tomamos o cuidado de evitar que um usuário acesse as contas de outros e que ele veja somente as dele, sempre!

- Por fim, partimos para mais duas *generic views*, usadas para a **alteração de senha**.

- Puxa, Alatazan, a cada dia que passa eu fico mais entusiasmada com o Django... olha só essas outras *generic views* que tem na contrib **"auth"**:

- django.contrib.auth.views.logout_then_login

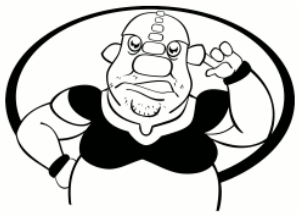
- django.contrib.auth.views.password_reset

- django.contrib.auth.views.redirect_to_login

- Só de ler os nomes, já dá pra ter uma ideia do que elas fazem, né...
- É, e olha: na verdade o Django tem muitas coisas que não vemos aqui, mas um passeio pela documentação é sempre muito excitante... outro dia, vi umas coisas sobre programação orientadas a testes por lá... é muito, muito maneiro!

No próximo capítulo, vamos conhecer um pouco mais disso: **TDD**, a metodologia de programação dirigida a testes aplicada no Django!

Capítulo 28: Programando com testes automatizados



A **Biblioteca de Explicações Inquestionáveis** de Katara fica um tanto bom ao sul das **Montanhas Fusoé**. Nesta biblioteca foram encontradas algumas das explicações mais profundas para perguntas não muito rasas ao longo dos últimos milênios.

Próximo dali está um igualmente milenar mosteiro, onde ficaram reclusos alguns dos maiores questionadores da História, por sua própria vontade ou não. Deste mosteiro surgiram as perguntas mais surpreendentes e improváveis dos últimos milênios.

À medida que os últimos milênios se passaram, tornara-se cada vez mais irrefutável a reputação da sabedoria inquestionável exalada deste complexo.

Entre o mosteiro e a biblioteca há uma plantação de um arbusto verde-bandeira, com folhas que em geral possuem 5 pontas grandes e outras pontas menores, chamadas de **"folha das 5 sabedorias e outras coisas que são quase isso"**.

O **Ritual da Apresentação das Respostas**, é um disputado evento anual que acontece na biblioteca, algumas semanas após o **Ritual do Desdobramento das Perguntas** do mosteiro. O restante do ano é preenchido com atividades comuns.

O Ritual da Apresentação das Respostas é precedido por um momento de pensamento e alegria profundos, quando são oferecidos chás, bolos e cigarros produzidos com as plantas da região. Após a apresentação das respostas, todos os presentes saem dali com a certeza de que mais algumas perguntas foram esclarecidas, de maneira enfática e inquestionável.

Uma das primeiras perguntas resolvidas e devidamente registrada na biblioteca fala da origem do universo conhecido e de seus planetas:

"Há milhares de anos, milhões de anos, milhares de milhões de anos, não haviam planetas, mas apenas uma fábrica de bolas de tênis. **Ogros** *trabalhavam ali, fazendo as mais criativas bolas de tênis do universo conhecido.*

Uma bola de tênis, para atingir o grau divino de qualidade, necessitava ser construída com cera de ouvido e água, tornando-se uma mistura lamacenta ao redor de uma estrutura consistente e resistente às mais diversas raquetadas. Elas devem resistir ao calor, ao frio, ao movimento circular sem ficarem tontas e a colisões com outras bolas também. Para dar uma emoção um pouco maior ao esporte, é preciso que as bolas possuam uma superfície irregular e seja costurada com linhas de tal modo que entre uma raquetada e outra nunca se sabe com certeza para qual direção a bola será arremessada.

A costura deve ser feita também de forma aleatória, para dificultar a infração dos direitos autorais e patentes da fabricação de bolas.

Os **ogros fazedores de bolas** *trabalhavam dia por dia, tempo por tempo e após cada bola concluída, davam-lhe uma raquetada e ela sumia para os confins do universo e passava a girar aleatoriamente em torno de alguma estrela, e por ali ficava."*

Dizem que isso também inspirou o surgimento dos primeiros protótipos e dos jogos com bolas.

Mais testes e menos debug

Por mais bizarro que seja um caso, é possível descrever um cenário de testes para ele. Por mais complexo que seja, é possível separar parte por parte em testes automatizados até que haja um conjunto satisfatório.

TDD - Test Driven Development (ou *Design*) - é uma metodologia de desenvolvimento de software que se aplica com essa filosofia: **um software deve ser escrito para atender a um conjunto de testes automatizados, que devem ser melhorados e detalhados à medida que este software evolui**.

Isso quer dizer que você deve escrever alguns testes automatizados, codificar para atender a esses testes, escrever mais testes (ou detalhar os já existentes) e mais software, e seguir com esse ciclo de vida.

Mas como assim? Como testar algo que ainda não existe?

Algumas das motivações mais relevantes para esse paradigma são:

1. A documentação de regras de negócio de um software nunca é clara o suficiente até atingir uma dimensão impossível de se acompanhar. Há diversos casos onde se escreve 300 ou 500 páginas para detalhar a análise de um software e o resultado é que ninguém é louco o suficiente para tentar ler essa quantidade de documentação a ponto de resolver ou compreender algo no software;

2. Quando um software recebe melhorias e evolui naturalmente, a cada versão lançada seria necessário o seu teste por completo, isso se torna inviável quando as versões são lançadas semanal ou mensalmente. A consequência é que não se testa por completo e ocorrem muitas ocasiões do tipo da *colcha curta*: **conserta de um lado e estraga do outro**;

3. Enquanto se escreve um conjunto de testes, muitas questões são esclarecidas ali, e a programação se torna mais limpa e o retrabalho diminui. Isso é porque você precisa pensar para escrever os testes, e isso o ajuda por exemplo a perceber que uma ideia *"brilhante"* na verdade era uma **bomba relógio**;

4. **Depurar sofware** (especialmente para a Web) **é um** *saco*, e nem sempre a depuração será feita com a perfeição e o roteiro ideais.

E assim por diante, enfim: depois que você pega o jeito, escrever testes se torna um investimento, e quanto melhor testado é seu software, maiores as chances de sucesso.

Como os testes se aplicam no Django

Há diversas formas de se escrever testes automatizados. No Django há pelo menos 4:

1. **Testes de unidade** - são usados para testar procedimentos isolados do software (sabe aquela fórmula de cálculo de um certo imposto?);

2. **DocTests textuais** - são usados para se testar **comportamentos** ou **roteiros** mais elaborados;

3. **DocTests em** *docstrings* - permitem que testes localizados sejam escritos nos próprios comentários de classes de modelo e seus métodos;

4. **Testes externos** - há frameworks como **Selenium** e **Twill** para testes de JavaScript e (X)HTML. Há também como se criar *scripts* para fazer **testes de performance** e *stress*, dentre outros meios para testar um software.

Para colocar alguns desses modos em prática, vamos criar uma nova aplicação, chamada **"mordomo"**. Portanto, crie a pasta **"mordomo"** dentro da pasta do projeto e dentro dela crie dois arquivos vazios, chamados **"__init__.py"** e **"models.py"**.

Abra o arquivo **"settings.py"** e localize a *setting* **INSTALLED_APPS**, acrescentando a nova aplicação a ela, para ficar assim:

```
INSTALLED_APPS = (
    'django.contrib.auth',
```

```
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.syndication',
    'django.contrib.flatpages',
    'django.contrib.comments',

    'blog',
    'galeria',
    'tags',
    'contas',
    'mordomo',
)
```

Salve o arquivo. Feche o arquivo.

Programando testes unitários

Testes unitários são testes de procedimentos que podem ser testados isoladamente. O Django possui um bom suporte para esse tipo de teste, através da classe **"django.test.TestCase"**.

Antes de mais nada, é preciso saber qual é a função dessa nova aplicação que estamos criando. A aplicação **"mordomo"** vai ter o papel de enviar notificações sobre o **estado do site** para os administradores do site e enviar também os **avisos de contas** que estão prestes a vencer.

Então, o que vamos fazer agora é criar um novo arquivo na pasta da aplicação **"mordomo"**, chamado **"tests.py"**, com o seguinte código dentro:

```
from django.test import TestCase


class TesteMordomo(TestCase):
    def testUmMaisUm(self):
        self.assertEquals(1 + 1, 3)
```

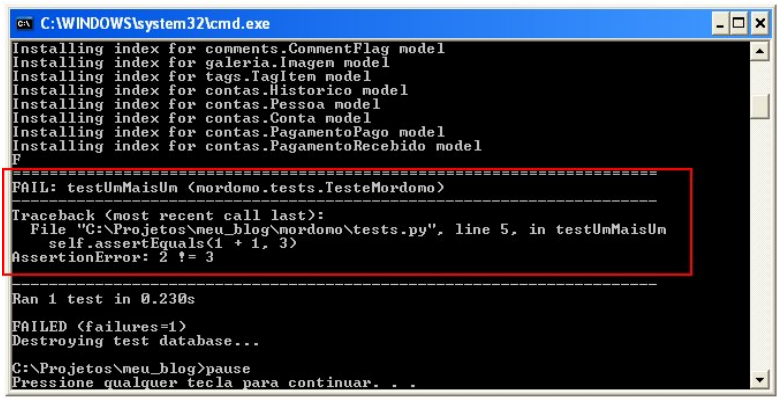
Salve o arquivo.

Agora na pasta do projeto, crie um novo arquivo chamado

"testar_mordomo.bat" com o seguinte código dentro:

```
python manage.py test mordomo
pause
```

Salve o arquivo. Feche o arquivo. Agora clique duas vezes sobre ele para executar o primeiro teste na nova aplicação:



Veja que o teste acusou um erro: **1 + 1 não é igual a 3** nem na Terra nem em lugar nenhum do universo que conhecemos.

Você pode notar que antes de executar o teste que escrevemos, todo o banco de dados foi gerado novamente. Isso ocorre porque de fato, ao executar os testes em um projeto, um banco de dados temporário é gerado para que os testes sejam feitos ali. Dessa forma, esse banco de dados é iniciado limpo e sem registros, exceto aqueles que foram definidos para serem gerados junto com a criação do banco.

Que tal melhorar um pouco mais os nossos testes?

Volte a editar o arquivo **"tests.py"** da pasta da aplicação **"mordomo"** e o modifique para ficar assim:

```
import datetime


from django.test import TestCase
from django.contrib.auth.models import User


from contas.models import Pessoa, Historico, ContaPagar, \
    CONTA_STATUS_APAGAR, CONTA_STATUS_PAGO
```

```
class TesteMordomo(TestCase):
    def setUp(self):
        self.usuario, novo = User.objects.get_or_create(
            username='admin'
            )

        self.pessoa, novo = Pessoa.objects.get_or_create(
            usuario=self.usuario, nome='Maria Anita',
            )

        self.historico, novo = Historico.objects.get_or_create(
            usuario=self.usuario, descricao='Agua',
            )

    def tearDown(self):
        pass

    def testUmMaisUm(self):
        self.assertEquals(1 + 1, 2)

    def testObjetosCriados(self):
        self.assertEquals(User.objects.count(), 1)
        self.assertEquals(Pessoa.objects.count(), 1)
        self.assertEquals(Historico.objects.count(), 1)
```

Puxa, mas quantas mudanças! Então, vamos entender o que fizemos ali...

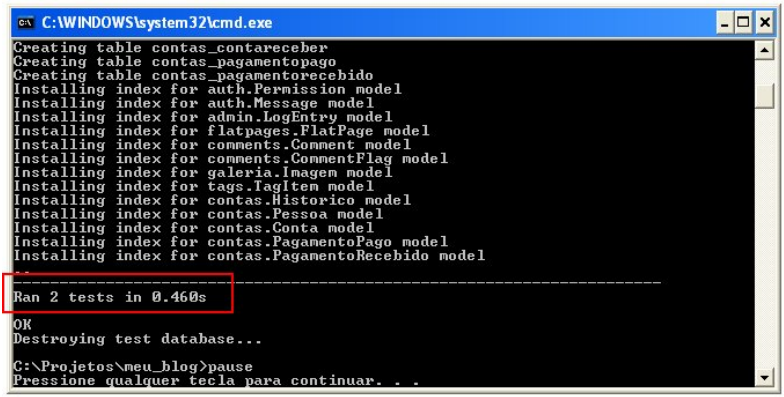
A classe que definimos possui 2 métodos de teste:

- testUmMaisUm
- testObjetosCriados

Antes de executar cada um dos métodos de teste, o método **"setUp"** é chamado, para preparar o terreno em que os testes irão trabalhar, após cada um deles, o método **"tearDown"** é executado para finalizar a bagunça feita, se necessário.

No caso acima, o teste **"testUmMaisUm"** não faz nada demais, e logo será removido, mas o teste **"testObjetosCriados"** verifica se os objetos básicos que criamos foram gravados com segurança no banco de dados, o que já é um passo na garantia de que algo está funcionando.

Salve o arquivo. Feche o arquivo. Agora clique duas vezes sobre o arquivo **"testar_mordomo.bat"** da pasta do projeto e veja o resultado:



Observe que uma parte destacada ali diz:

```
Ran 2 tests in 0.460s
```

E logo em seguida há outra linha que o teste fechou OK:

```
OK

Destroying test database...
```

Escrevendo testes para depois programar

E agora que já temos uma visão geral de como funcionam os testes, vamos colocar a ideia da programação orientada a testes. Abra novamente o arquivo **"tests.py"** da pasta da aplicação **"mordomo"** para edição e acrescente este novo método de teste ao final:

```
    def testContasPendentes(self):
        from mordomo.models import obter_contas_pendentes


        for numero in range(-50,50):
            if numero % 2:
                novo_status = CONTA_STATUS_APAGAR
```

```
            else:
                novo_status = CONTA_STATUS_PAGO

            nova_data = datetime.date.today() + \
    datetime.timedelta(days=numero)

            novo_valor = numero + 70

            ContaPagar.objects.create(
                usuario=self.usuario,
                pessoa=self.pessoa,
                historico=self.historico,
                data_vencimento=nova_data,
                valor=novo_valor,
                status=novo_status,
                )

        self.assertEquals(ContaPagar.objects.count(), 100)

        contas_pendentes = obter_contas_pendentes(
            usuario=self.usuario,
            prazo_dias=10,
            )

        self.assertEquals(contas_pendentes.count(), 30)
```

Novamente, temos um código longo. Vamos ver detalhe por detalhe?

A primeira coisa que será verificada será esta importação:

```
        from mordomo.models import obter_contas_pendentes
```

Se não existe a função **"obter_contas_pendentes"**, seguramente uma exceção do tipo **"ImportError"** será levantada antes que todo o restante seja chamado.

A próxima parte do código inicia um laço de **100 posições** entre os números -50 e 50 (que de fato pelos padrões do Python será de -50 a 49) para criar 100 contas a pagar:

```
        for numero in range(-50,50):
```

Os números vão de -50 a 49 pois cada uma das contas terá um **status**, **data de vencimento** e **valor** diferentes, entre contas vencidas e por vencer.

O próximo trecho do código escolhe um dos **status** que indicam se a conta já está paga ou ainda está pendente:

```
            if numero % 2:
                novo_status = CONTA_STATUS_APAGAR
            else:
                novo_status = CONTA_STATUS_PAGO
```

Na prática, o código acima indica o status **"CONTA_STATUS_APAGAR"** para quando **"numero"** for um valor ímpar e o status **"CONTA_STATUS_PAGO"** para quando ele for par. A operação **"numero % 2"** calcula o módulo **2** sobre o valor de **"numero"** que será sempre **1 (ímpar) ou 0 (par)**.

A próxima linha calcula uma data que para cada conta será incrementada em um dia:

```
            nova_data = datetime.date.today() + \
    datetime.timedelta(days=numero)
```

Na prática, estamos calculando a data atual somada à quantidade de dias apontada pela variável **"numero"**, que vai de **-50** (50 dias atrás) a **49** (daqui a 49 dias), ou seja: metade das contas estarão vencidas e outra metade ainda a vencer.

A próxima linha calcula um valor que também é incrementado para cada conta:

```
            novo_valor = numero + 70
```

Na primeira conta, o valor será **"-50 + 70"**, ou seja: **20**. Na segunda ela será **21** e assim por diante.

O próximo trecho de código cria a conta a pagar:

```
            ContaPagar.objects.create(
                usuario=self.usuario,
                pessoa=self.pessoa,
                historico=self.historico,
                data_vencimento=nova_data,
                valor=novo_valor,
                status=novo_status,
                )
```

Ela é criada usando os valores que calculamos e os objetos **"self.usuario"**, **"self.pessoa"** e **"self.historico"**.

Após as 100 contas serem criadas, fazemos uma pequena verificação:

```
        self.assertEquals(ContaPagar.objects.count(), 100)
```

A comparação acima serve apenas para confirmar que as 100 contas foram criadas com sucesso.

A próxima etapa faz uso da função que importamos no início do método:

```
        contas_pendentes = obter_contas_pendentes(
            usuario=self.usuario,
            prazo_dias=10,
            )
```

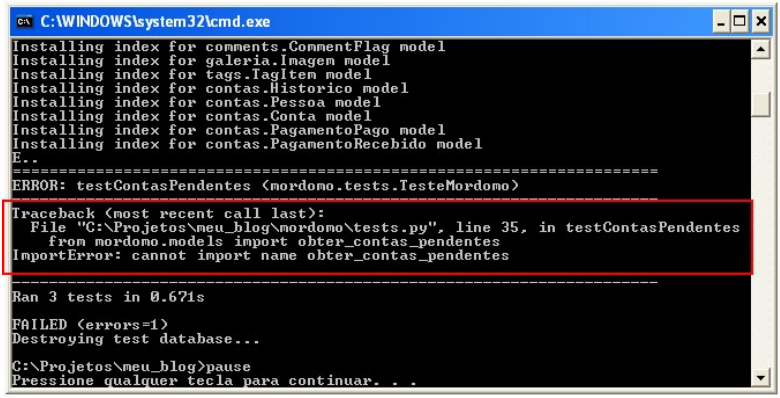
Ela será chamada para um usuário específico e com um prazo máximo em dias após a data atual. Lembre-se de que o mordomo deve avisar ao usuário das contas a vencer, e não somente daquelas que já venceram. **10 dias** é um prazo bacana.

Por fim, a lista de contas pendentes resultantes da chamada à função também é verificada:

```
        self.assertEquals(contas_pendentes.count(), 30)
```

Veja que temos 100 contas, das quais **60 delas** venceram antes da data atual ou vencerão até os próximos **10 dias**. Como somente a metade possui status **"a pagar"**, a quantidade de contas pendentes encontradas deve ser **igual a 30**.

Salve o arquivo. Feche o arquivo. Que tal agora executar nosso teste? Clique duas vezes sobre o arquivo **"testar_mordomo.bat"** da pasta do projeto e veja o resultado:



Como você já sabia, uma exceção de importação foi levantada, já que a função não existe. Pois então o primeiro passo na nossa nova forma de programar é resolver esse primeiro erro **criando a função**.

Na pasta da aplicação **"mordomo"**, abra o arquivo **"models.py"** para edição e escreva o seguinte código dentro:

```
import datetime


from django.contrib.auth.models import User


from contas.models import Conta, CONTA_STATUS_APAGAR


def obter_contas_pendentes(usuario, prazo_dias):
    delta = datetime.timedelta(days=prazo_dias)


    return Conta.objects.filter(
        usuario=usuario,
        status=CONTA_STATUS_APAGAR,
        data_vencimento__lte=datetime.date.today() + delta
        )
```

Nas primeiras linhas do arquivo, importamos os módulos necessários e declaramos a função:

```
import datetime


from django.contrib.auth.models import User


from contas.models import Conta, CONTA_STATUS_APAGAR


def obter_contas_pendentes(usuario, prazo_dias):
```

Logo em seguida definimos uma variável chamada **"delta"** para armazenar o objeto que representa uma faixa de data para a quantidade de dias informada no argumento **"prazo_dias"**:

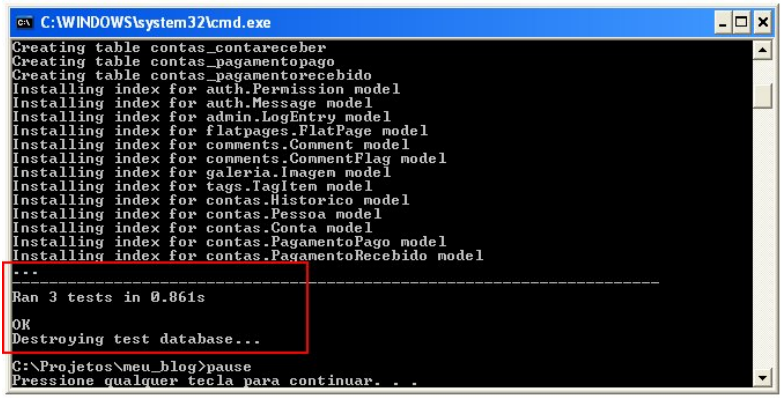
```
    delta = datetime.timedelta(days=prazo_dias)
```

E finalmente a função retorna as contas do **usuário informado**, com status **"a**

pagar" e data de vencimento **menor ou igual** à data atual, somada do prazo de dias representado na variável **"delta"**:

```
    return Conta.objects.filter(
        usuario=usuario,
        status=CONTA_STATUS_APAGAR,
        data_vencimento__lte=datetime.date.today() + delta
        )
```

Salve o arquivo. Feche o arquivo. Execute o teste novamente e veja agora o resultado:



Bacana, agora está tudo certo!

Testando e-mails enviados

Agora que temos uma função para carregar as contas pendentes para um usuário específico, precisamos criar outra para enviar a ele uma lista delas por e-mail.

Para isso, vamos novamente começar criando testes. Abra o arquivo **"tests.py"** da pasta da aplicação **"mordomo"** para edição e localize esta linha:

```
        from mordomo.models import obter_contas_pendentes
```

Modifique-a para ficar assim:

```
        from mordomo.models import obter_contas_pendentes, \
    enviar_contas_pendentes
```

Agora acrescente o seguinte trecho de código ao final do arquivo, mas ainda dentro do método **"testContasPendentes"**:

```
        enviar_contas_pendentes(
            usuario=self.usuario,
            prazo_dias=10,
            )

        from django.core import mail

        self.assertEquals(len(mail.outbox), 1)
        self.assertEquals(
        mail.outbox[0].subject, 'Contas a pagar e receber'
        )
```

Simplificando o que as novas linhas de código fazem...

A primeira parte chama uma nova função, que envia ao usuário a lista das contas pendentes para seu e-mail:

```
        enviar_contas_pendentes(
            usuario=self.usuario,
            prazo_dias=10,
            )
```

Já na segunda parte, usamos o pacote **"django.core.mail"** para verificar se a caixa de mensagens (**"mail.outbox"**) possui **uma mensagem** e em seguida, verificamos se o título dela é **"Contas a pagar e receber"**:

```
        from django.core import mail

        self.assertEquals(len(mail.outbox), 1)
        self.assertEquals(
        mail.outbox[0].subject, 'Contas a pagar e receber'
        )
```

Salve o arquivo. Feche o arquivo. Você pode executar os testes, mas o que vai verificar é que deve ser escrito o código para atender às novas especificações que ele aponta.

Para, isso, abra novamente o arquivo **"models.py"** da aplicação **"mordomo"** para edição e acrescente as seguintes linhas de código ao final:

```
def enviar_contas_pendentes(usuario, prazo_dias):
```

```
    contas_pendentes = obter_contas_pendentes(usuario,
prazo_dias)


    if not contas_pendentes.count():
        return False


    texto = """
    Voce tem as seguintes contas vencidas ou a vencer nos
proximos %(dias)d dias:


    %(lista_de_contas)s
    """%{
        'dias': prazo_dias,
        'lista_de_contas': "\n".join(
        [unicode(conta) for conta in contas_pendentes]
        ),
        }


    return usuario.email_user(
        'Contas a pagar e receber',
        texto,
        )
```

Veja que nas primeiras duas linhas, nós declaramos a nova função e fazemos uso da função **"contas_pendentes"**:

```
def enviar_contas_pendentes(usuario, prazo_dias):
    contas_pendentes = obter_contas_pendentes(
    usuario,
    prazo_dias
    )
```

Em seguida, saímos da função caso não existam contas pendentes:

```
    if not contas_pendentes.count():
        return False
```

Definimos o **texto da mensagem**, formatando a *string* indicando os dias e a

lista de contas separadas por saltos de linha (**"\n"**):

```
    texto = """
    Voce tem as seguintes contas vencidas ou a vencer nos
proximos %(dias)d dias:


    %(lista_de_contas)s
    """%{
        'dias': prazo_dias,
        'lista_de_contas': "\n".join(
        [unicode(conta) for conta in contas_pendentes]
        ),
        }
```

E por fim, a função retorna a quantidade de e-mails enviados pelo método **"email_user"** do usuário, que envia um mensagem para o endereço de e-mail do usuário em questão:

```
    return usuario.email_user(
        'Contas a pagar e receber',
        texto,
        )
```

O primeiro argumento indica o título (ou *subject*) da mensagem e o segundo argumento indica seu **texto**.

Salve o arquivo. Feche o arquivo.

Agora, a resposta para sua pergunta mais profunda é: e-mails enviados durante a execução de testes são falsos. É... é verdade! Tudo funciona direitinho como se fosse de verdade, mas o que acontece é que eles são armazenados na variável **"outbox"** do pacote **"django.core.mail"**. As mesmas rotinas de e-mail funcionam corretamente quando o sistema estiver executando normalmente, fora do ambiente de testes.

Trabalhando com DocTests

Testes unitários são legais, divertidos até, possuem muitos recursos... mas, são de fato, limitados.

Cada um daqueles métodos de testes deve ser encarado e pensado isoladamente, pois sua execução não segue uma ordem certa... como seus próprios nomes dizem:

são **testes unitários**.

Para testar cenários mais complexos e compostos de várias etapas ou elementos - geralmente chamados de **testes de comportamento** - existem os **DocTests**. Os testes de comportamento, em um formato um pouco mais elaborado são conhecidos como **BDD - "Programação Orientada a Comportamentos"**.

Para conhecer um pouquinho de **DocTests**, abra o arquivo **"tests.py"** da aplicação **"mordomo"** para edição e acrescente as seguintes linhas de código ao início do arquivo, empurrando todo o restante para baixo:

```
"""
Criar um comando para verificar se estah tudo ok na pagina
principal do site e enviar um e-mail caso algum erro foi
encontrado.


    >>> from mordomo.models import verificar_pagina_inicial


Simulando status 200 - estado ok


    >>> def status_code_ok(url):
    ...     return 200


    >>> verificar_pagina_inicial(
    ... funcao_status_code=status_code_ok
    ... )
    True
"""
```

Veja que estamos trabalhando numa grande *string*. Logo de primeira já dá pra notar o quanto o DocTest é mais amigável, pois ele permite textos livres explicando detalhadamente o que está acontecendo ali, como se fosse uma história sendo contada.

As linhas iniciadas com **quatro espaços**, seguidas de três sinais de **"maior-quê"** e mais um espaço, iniciam um procedimento em Python. Caso esses procedimentos necessitem de mais linhas (como é um caso de uma função ou classe, por exemplo), então as linhas secundárias devem estar exatamente abaixo das anteriores, iniciadas sempre com **quatro espaços**, **três pontos** e mais um espaço, seguidas da identação comum à linguagem.

Já as linhas iniciadas com **quatro espaços** e seguidas de qualquer outra coisa, indicam que aquele é o valor esperado como resposta da linha de comando anterior, veja este caso por exemplo:

```
    >>> verificar_pagina_inicial(
    ... funcao_status_code=status_code_ok
    ... )
    True
```

A linha com **" True"** indica que a função chamada na linha anterior deve retornar o valor **"True"**.

Viu como é simples?

O que fizemos aí foi uma função, chamada **"status_code_ok"**, que retorna forçadamente o código **200**, que é o código de status HTTP que significa que tudo está OK. Outros códigos são o **404** - que indica **"página não encontrada"** -, **500** - que indica **"erro no servidor"**-, e há muitos outros além desses.

Agora acrescente ao final do DocTest (antes da linha que finaliza com três aspas duplas, assim: **"""**) mais este trecho de código:

```
    >>> from django.core import mail
    >>> len(mail.outbox)
    0


Simulando status 500 - estado de erro que envia e-mail


    >>> def status_code_500(url):
    ...     return 500


    >>> verificar_pagina_inicial(
    ... funcao_status_code=status_code_500
    ... )
    True


    >>> len(mail.outbox)
    1
```

O primeiro trecho de código verifica a quantidade de mensagens na caixa de e-mails enviados, que deve ser zero:

```
>>> from django.core import mail
>>> len(mail.outbox)
0
```

Lembre-se de que, como diz o próprio comentário no DocTest que escrevemos, a função **"verificar_pagina_inicial"** deve enviar um e-mail de aviso somente se algo aconteceu de errado com a página inicial.

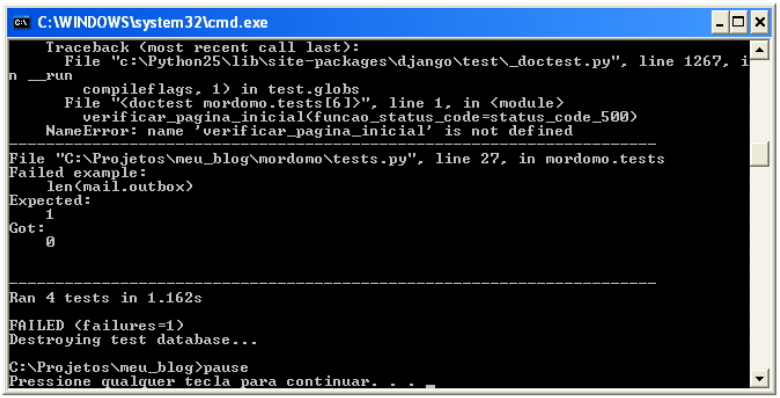
No segundo trecho de código, nós criamos outra função para forçar o código de status **500**, que indica erro na página:

```
>>> def status_code_500(url):
...     return 500


>>> verificar_pagina_inicial(
... funcao_status_code=status_code_500
... )
True
```

E por fim o último trecho espera que após este erro simulado haja um e-mail enviado na caixa:

```
>>> len(mail.outbox)
1
```

Salve o arquivo. Feche o arquivo. Execute os testes, clicando duas vezes sobre o arquivo **"testar_mordomo.bat"** para ver o resultado das nossas mudanças:



Como pode ver, o nosso DocTest levantou uma fila de erros, mas todos eles foram resumidos como se fossem apenas um, por estas linhas, veja:

```
Ran 4 tests in 1.162s


FAILED (failures=1)
```

Note que se antes nós tínhamos 3 testes e agora temos 4, é porque o DocTest é interpretado pelo Django com um único teste, e é por isso que o sumário final aponta apenas **"1 falha"** ("*failures=1*").

Que tal agora escrever o código que vai acalmar esses testes e voltá-los ao estado de **"tudo OK"**?

Abra o arquivo **"models.py"** da aplicação **"mordomo"** para edição e acrescente as seguintes linhas de código ao final:

```
from django.core.mail import send_mail


def verificar_pagina_inicial(funcao_status_code):
    status_code = funcao_status_code('http://localhost:8000/')


    if status_code != 200:
        send_mail(
            'Erro na pagina inicial do site!',
            'Ocorreu um erro no site, verifique!',
            'alatazan@gmail.com',
            ['alatazan@gmail.com'],
            )


    return True
```

Veja que em primeiro de tudo nós importamos a função de envio de e-mails do Django: **"send_mail"**:

```
from django.core.mail import send_mail
```

Depois disso a declaramos, com o argumento que vai receber a função que verifica o código de status da URL, e essa função é executada, recebendo a URL da página inicial:

```
def verificar_pagina_inicial(funcao_status_code):
    status_code = funcao_status_code('http://localhost:8000/')
```

A seguir, o código de status é comparado com **200**, o único valor válido esperado. Se ele for diferente, temos um erro na página, e então o e-mail é enviado

ao endereço de e-mail **"alatazan@gmail.com"**:

```
    if status_code != 200:
        send_mail(
            subject='Erro na pagina inicial do site!',
            message='Ocorreu um erro no site, verifique!',
            from_email='site_do_alatazan@gmail.com',
            recipient_list=['alatazan@gmail.com'],
            )
```

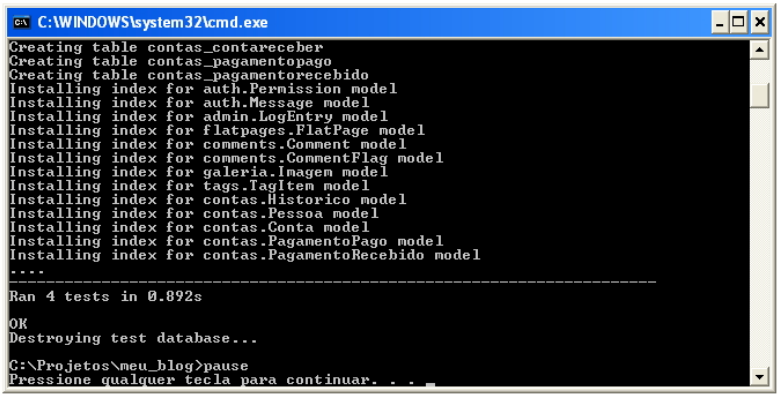
Por fim, o retorno esperado **"True"** é retornado.

Observando bem, você vai notar que usando DocTests nós criamos um teste mais humano e amigável, que segue uma sequência de ações para representar um cenário esperado de um comportamento do sistema.

Criando as funções que forçam o código do status da URL nós criamos situações falsas para que seja possível testar as diversas situações pelas quais o sistema pode passar. Com um pouco de prática e imaginação, os DocTests permitem a você uma variedade de testes quase infinita, mas isso, só depende de você!

Salve o arquivo. Feche o arquivo.

Execute o teste da aplicação, clicando duas vezes sobre o arquivo **"testar_mordomo.bat"** e veja o resultado:



Você deve estar se perguntando quando essas funções serão usadas fora do ambiente de testes... mas isso é para o próximo capítulo!

Usando as funções do mordomo em uma ferramenta para linha de comando

Naquela noite, Alatazan refletiu sobre isso tudo, e viu que escrever os testes antes da codificação faz todo o sentido:

- Com testes bem escritos, diminui a necessidade de descrever documentação sobre o código, também diminui as fases de análise e arquitetura. Um teste bem escrito vale por tudo isso e ainda garante que o software continue funcionando...

Basicamente o que Alatazan fez hoje foi:

- Criar uma nova aplicação vazia e iniciar um arquivo chamado **"tests.py"** dentro dela;

- O arquivo **"tests.py"** também poderia ser uma pasta **"tests"**, com um arquivo **"__init__.py"** dentro dela;

- Testes unitários são testes para calcular procedimentos isolados e independentes;

- Usando os métodos de **assertação** - todos aqueles que começam com **"assert..."** - é possível verificar os valores da forma como se espera que eles sejam. Há métodos de assertação também para verificar valores semelhantes (ainda que não sejam iguais), valores verdadeiros ou falsos, exceções levantadas, validação de formulários, estados de URLs e outras coisas mais;

- Uma classe de testes deve herdar da classe **"django.test.TestCase"**;

- O método **"setUp"** é executado antes de cada um dos métodos de teste daquela classe de testes;

- O método **"tearDown"** é o contrário do **"setUp"**, pois é executado depois de cada um dos métodos de teste da classe;

- Nunca escreva testes longos antes de programar. Escreva de meia-dúzia a dez testes, codifique até cada um deles ser atendido, depois escreva mais de meia-dúzia a dez, e aos poucos vá quebrando esses testes em outros menores, cada vez mais detalhados e claros, e codificando para se ajustar a isso;

- Faça alguns testes com maiores quantidades de dados para verificar como a aplicação se comporta;

- **DocTests** são mais "rústicos" que testes unitários, mas têm mais poder, pois é possível testar cenários e comportamentos complexos;

- Os e-mails enviados durante a execução dos testes são falsos: eles são apenas armazenados na variável **"outbox"** do pacote **"django.core.mail"**;

- Escreva seu código sempre de forma flexível, aceitando funções para fazer determinadas rotinas de forma que possam ser substituídas por **funções falsas**, assim você pode testar com segurança e ainda mantém seu código fácil de se adaptar a futuras novidades;

- A *string* de um DocTest deve ser escrita nas primeiras linhas do arquivo.

E pensando nisso, Alatazan dormiu... no dia seguinte o desafio seria criar os chamados **"management commands"**, que são recursos para se criar ferramentas executáveis na linha de comando.

Capítulo 29: Criando ferramentas para a linha de comando

Alatazan estava em êxtase!

Esse era o último dia que estudava Django na modalidade básica junto com seus amigos. Nos últimos dias, Cartola e Nena estiveram distantes, ora viajando, ora envolvidos em seus trabalhos de final de ano, mas Alatazan estava ali firme e queria resolver uma última coisa antes de entrar de férias: executar algumas coisas do site a partir de um *prompt* de linhas de comando.

O desafio de hoje - o derradeiro desse estudo - mostra que é absurdamente simples criar esse tipo de *script* no Django. Muito mais simples do que tentar reinventar a roda, com certeza.

Tenha um bom estudo e aproveite com a mesma paixão que Alatazan estudou nesses últimos meses!

Ferramentas para a linha de comando

Às vezes é necessário criar uma ferramenta para ser chamada pela linha de comando. Algumas vezes é preciso criar esse tipo de ferramenta para automatizar tarefas, e outras vezes para facilitar a execução de certas rotinas que são feitas manualmente de vezes em quando.

A nossa aplicação **"mordomo"** foi criada para fazer algumas dessas tarefas, especialmente duas:

1. Verificar se a página principal do site está OK e enviar um e-mail caso algo esteja dando errado;
2. Enviar e-mails periódicos aos usuários que possuam contas a pagar em até 10 dias contando a partir da data atual.

Criando um primeiro *script*

A reação natural a essa situação é criar um *script* externo ao ambiente do Django e tentar resolver tudo por conta própria, mas o resultado geralmente é um conjunto de pequenas gambiarras, isso quando não se desiste antes da hora.

O que muita gente não sabe é que o Django tem recursos especificamente para isso.

Quando você executa um comando como este:

```
python manage.py syncdb
```

Ou este:

```
python manage.py dumpdata contas > contas.json
```

Mal sabe que os comandos **"syncdb"** e **"dumpdata"** são arquivos que seguem uma API simples do Django, criada para isso. Então, você pode ter seus próprios **"python manage.py verificar_status"** ou **"python manage.py enviar_contas_por_email"** sem nenhuma dificuldade!

A esse tipo de recurso, usamos o nome de **"Management Commands"**.

Criando o primeiro comando

Na pasta da aplicação **"mordomo"**, crie uma pasta chamada **"management"** e dentro dela um arquivo vazio, chamado **"__init__.py"**. Dentro da nova pasta, crie outra, chamada **"commands"** e dentro dela outro arquivo, também vazio, também chamado **"__init__.py"**.

Agora dentro da pasta **"commands"** crie um arquivo chamado **"verificar_status.py"**, com o seguinte código dentro:

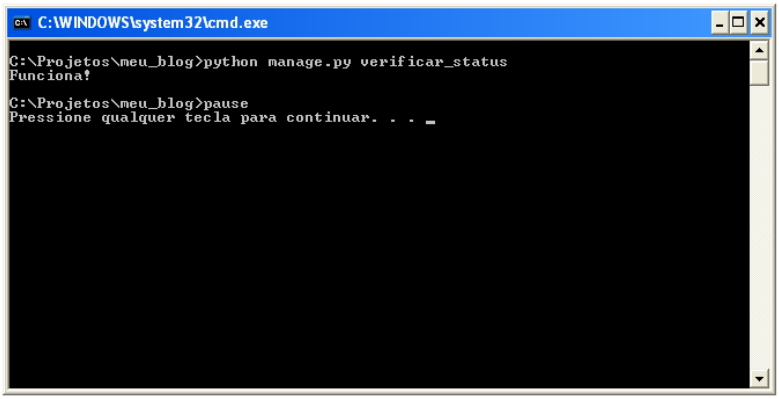
```
from django.core.management import BaseCommand


class Command(BaseCommand):
    def handle(self, **kwargs):
        print 'Funciona!'
```

Salve o arquivo.

Agora na pasta do projeto, crie um novo arquivo chamado **"verificar_status.bat"** com o seguinte código dentro:

```
python manage.py verificar_status
pause
```

Salve o arquivo. Feche o arquivo. Clique duas vezes sobre o novo arquivo para executá-lo e veja o resultado:



Viu como é fácil? Agora vamos dar mais um passo. Na pasta da aplicação **"mordomo"**, abra o arquivo **"models.py"** para edição e localize estas linhas de código:

```
def verificar_pagina_inicial(funcao_status_code):
    status_code = funcao_status_code('http://localhost:8000/')
```

Nós temos aí dois problemas:

1. O argumento **"funcao_status_code"** não possui um valor padrão, o que faz esta função ser dependente de uma função ser atribuída externamente sempre que ela for chamada;

2. A função está presa à URL **"http://localhost:8000/"**, o que faz dela limitada.

Para resolver o primeiro problema, escreva a seguinte função acima das linhas de código encontradas:

```
import urllib2


def obter_status_code(url):
    try:
        urllib2.urlopen(url)
    except urllib2.HTTPError, e:
        return e.code
    except Exception:
```

```
        return 0


    return 200
```

Esta função faz nada mais do que retornar o código de status da URL requisitada, que é exatamente do que precisamos. Ela tenta a conexão normal à URL. Se a conexão levantar qualquer erro do tipo **"urllib2.HTTPError"**, a função retorna o código de status do erro. Se for outro erro qualquer, ela retorna **0** (zero). E se der tudo certo, ela retorna **200**: o código de status que significa **OK**.

Agora modifique as seguintes linhas:

```
def verificar_pagina_inicial(funcao_status_code):
    status_code = funcao_status_code('http://localhost:8000/')
```

Para ficar assim:

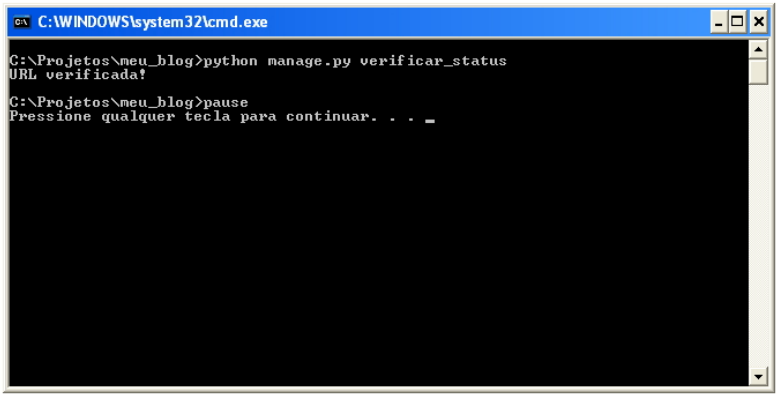
```
def verificar_pagina_inicial(
    funcao_status_code=obter_status_code,
    url='http://localhost:8000/',
    ):
    status_code = funcao_status_code(url)
```

Ou seja, indicamos a função que criamos como valor padrão para o argumento **"funcao_status_code"** e substituímos a URL pelo argumento **"url"**, que por sua vez possui como valor padrão a mesma URL que usávamos antes de forma fixa.

Salve o arquivo. Feche o arquivo. Agora volte a editar o arquivo **"verificar_status.py"** da pasta **"mordomo/management/commands"**, partindo da pasta do projeto e o modifique para ficar assim:

```
from django.core.management import BaseCommand
from mordomo.models import verificar_pagina_inicial


class Command(BaseCommand):
    def handle(self, **kwargs):
        verificar_pagina_inicial()
        print 'URL verificada!'
```

Salve o arquivo. Execute o arquivo **"verificar_status.bat"** da pasta do projeto novamente, para que ele funcione corretamente:



Trabalhando com opções

Agora, é hora de melhorar um pouco mais o nosso comando, adicionando uma opção importante: a URL a ser verificada.

Para isso, volte ao arquivo que acabamos de modificar (**"verificar_status.py"**) e o modifique novamente, para ficar assim:

```
from optparse import make_option


from django.core.management import BaseCommand
from django.conf import settings


from mordomo.models import verificar_pagina_inicial


class Command(BaseCommand):
    option_list = BaseCommand.option_list + (
        make_option('--url',
                    default='/',
                    dest='url',
                    help='Informe a URL para verificar o status',
            ),
        )
    help = \
```

```
    'Verifica uma URL e envia e-mail ao admin se estiver com erro'
    requires_model_validation = True

    def handle(self, url, **kwargs):
        url = settings.PROJECT_ROOT_URL[:-1] + url
        verificar_pagina_inicial(url=url)
        print 'URL "%s" verificada!'%url
```

Uau! Vamos detalhar o que o nosso novo código faz!

Primeiro, nossas importações aumentaram, agora usamos a biblioteca **"optparse"** do Python para ter o recurso de opções em nosso *script*. Também usamos o módulo **"settings"** do projeto para carregar dali a URL padrão do site:

```
from optparse import make_option


from django.core.management import BaseCommand
from django.conf import settings


from mordomo.models import verificar_pagina_inicial
```

Em seguida, acrescentamos uma nova opção além daquelas que o próprio Django tem por padrão em todas elas, que é a opção **"--url"**, que possui um valor *default*: **"/"**. O valor desta opção será concatenado à URL raiz do projeto (que por padrão é **"http://localhost:8000/"**):

```
class Command(BaseCommand):
    option_list = BaseCommand.option_list + (
        make_option('--url',
                    default='/',
                    dest='url',
                    help='Informe a URL para verificar o status',
            ),
        )
```

O atributo a seguir tem a função de exibir uma mensagem de ajuda sobre o comando:

```
    help = 'Verifica uma URL e envia e-mail ao admin se estiver
com erro'
```

E este outro exige que o Django faça uma validação nos arquivos de modelo

antes de permitir a execução do comando:

```
    requires_model_validation = True
```

Por fim, o método que executa a ação do comando foi modificado para receber o argumento **"url"** e concatená-lo à URL raiz do site:

```
    def handle(self, url, **kwargs):
        url = settings.PROJECT_ROOT_URL[:-1] + url
        verificar_pagina_inicial(url=url)
        print 'URL "%s" verificada!'%url
```

Salve o arquivo. Feche o arquivo. Agora, para finalizar a nossa mudança, abra o arquivo **"settings.py"** da pasta do projeto para edição e localize esta linha de código:

```
PROJECT_ROOT_PATH = os.path.dirname(os.path.abspath(__file__))
```

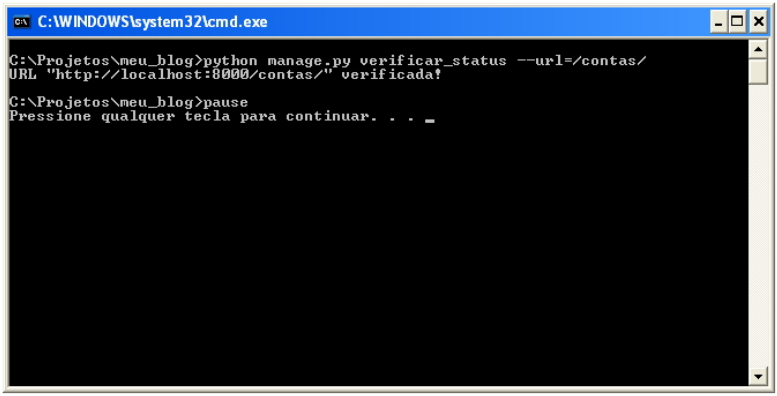
Acrescente esta abaixo dela:

```
PROJECT_ROOT_URL = 'http://localhost:8000/'
```

Salve o arquivo. Feche o arquivo. Agora para usar as mudanças que fizemos abra o arquivo **"verificar_status.bat"** da pasta do projeto para edição e o modifique para ficar assim:

```
python manage.py verificar_status --url=/contas/
pause
```

Salve o arquivo. Feche o arquivo. Execute esse arquivo e veja como ele se sai:



Fantástico! Temos uma ferramenta de comandos e foi mole, mole pra fazer!

Terminando o curso e partindo para novos desafios

Alatazan falou em voz alta, como que estivesse fazendo um gol:

- Em contagem regressiva, vamos ver o que vimos hoje... é tão pouco que vou ser breve:
    - O primeiro passo foi criar a pasta **"management/commands"** dentro da pasta da aplicação, com seus respectivos **"__init__.py"**;
    - Dentro da pasta, criamos um arquivo com o mesmo nome do comando que usei: **"verificar_status.py"**. É preciso um pouco de cuidado pra evitar criar comandos que já existem em outras aplicações com o mesmo nome;
    - O novo *script* possui uma classe chamada **"Command"**, um nome padrão, que estende da classe **"django.core.management.BaseCommand"** e declara um método **"handle"** para executar a ação do comando;
    - Para criar uma opção, usamos a biblioteca **"optparse"** e dela usamos a função **"make_option"**.
    - O atributo **"help"** é bacana para ajudar ao usuário a usar aquele comando e a opção em questão;
    - O atributo **"requires_model_validation"** determina que este comando depende de uma validação dos arquivos de modelo antes, isso porque os arquivos de modelo são importantes neste caso.
- E é só isso!

Alatazan recebeu um telefonema. Para ele foi um alívio danado saber que agora teria um emprego. O que ele não sabia ainda era que aquela oportunidade tinha um lado negro... e Nena sabe muito bem do que se trata um **emprego tradicional**.

Capítulo 30: Adentrando a selva e conhecendo os verdadeiros bichos

Uma inexplicável e irônica verdade é a de que principiantes realmente têm sorte.

O primeiro salto com para-quedas, a primeira volta de patins, o primeiro café e o primeiro telescópio que você faz - **todos funcionam melhor do que se esperava**. O desastre é reservado à segunda e terceira tentativas e assim por diante até a décima ou mais.

Há um ditado em Katara que diz que **"se você caiu na primeira tentativa, é porque não vai cair na décima sétima"**, devido a uma crença local de que as primeiras 17 tentativas, exceto à primeira, são desastrosas.

Num centro comercial encardido de uma rua não muito ensolarada, de muros e paredes também encardidos e vigiadores de carros igualmente encardidos, a única entrada desce uma escada e na terceira porta à direita fica a **Internet On-Line**, uma empresa de construção de sites focada em resolver todo tipo de peleja *on-line*.

Um cartaz distribuído por alguns dos postes da cidade diziam: *"Problemas no amor? Está difícil arrumar um emprego? Quer abrir uma loja na internet e vender pro mundo inteiro? Venha à Internet On-Line que temos uma solução para você."*

Os últimos 27 desenvolvedores que trabalharam na **I://OL** (sigla "bem-sacada", de acordo com quem teve a ideia) partiram para outra antes que o trabalho fosse terminado. O proprietário, diretor, vendedor e visionário da I://OL prefere ser chamado de **"SaradoMSNy"**, seu identificador de e-mail e em qualquer mensageiro instantâneo que se possa imaginar.

Alatazan acompanhou tudo isso seguindo a cada metro antes que batesse à porta, mas para quem é de outro planeta, não parece uma furada *tãão furada* assim.

SaradoMSNy descobriu o Django após algumas tentativas sem muito sucesso

com outras ferramentas, afinal, seus prazos de 30 dias nunca eram compatíveis com o que de fato acontecia e ele não sabia porque as coisas andavam assim tão complicadas, afinal, se fosse tão complicado assim, *"como Sergey Brin e Larry Jeans teriam aberto uma empresa de internet numa garagem?"*.

O primeiro desafio para Alatazan nem era tão complicado assim. Daí por diante, só **Zaron** poderia saber, mas Alatazan é otimista e não tem a menor ideia de quem é **Zaron**, e mais que qualquer outra coisa, é alheio à maioria das dificuldades conhecidas pelos terráqueos. Ele simplesmente quer colocar em prática as boas práticas que aprendeu, e que seja em Django!

Ao longo desses 30 capítulos, conhecemos o que o Django apresenta em seus fundamentos.

Se você cultiva o **conhecimento e as boas práticas**, você tem lugar garantido para se apaixonar pelo Django, mas se não é, há espaço pra você também.

Como toda ferramenta madura, há muito mais a conhecer, e há muitas ferramentas terceiras que podem ser aproveitadas para ganhar tempo e qualidade. A maioria dessas ferramentas são chamadas de **"aplicações plugáveis"**.

É disso que vamos tratar na sequência. Siga em frente, e conheça as várias aplicações práticas disponíveis para Django.